Compositional Verification using Assume - Guarantee Approach

by

P. Prahallada Reddy

TH
EE/2001/M
R246c

DEPARTMENT OF ELECTRICAL ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY, KANPUR

January, 2001

Compositional Verification using Assume - Guarantee Approach

A thesis Submitted
in Partial Fulfillment of the Requirements
for the Degree of

Master of Technology

by

P. Prahallada Reddy

to the

DEPARTMENT OF ELECTRICAL ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY KANPUR

JANUARY, 2001

CERTIFICATE

This is to certify that the work contained in this thesis entitled "**Compositional Verification using Assume - Guarantee Approach**" submitted by **P. Prahallada Reddy** in partial fulfillment of the requirements for the award of Master of Technology has been carried out under my supervision and guidance and that this work has not been submitted elsewhere for the award of a degree.

[illegible]

Date : January 16, 2001

(Dr. S. K. Roy)
Department of Electrical Engineering
Indian Institute of Technology
Kanpur - 208016

ACKNOWLEDGEMENTS

I am indebted to my thesis advisor, **Dr. S. K. Roy,** for introducing me to this fascinatıng field of formal verification. I express my deepest gratitude to him for providing me all the support, patiently listening to my problems and providing valuable suggestions.

I am grateful to my teachers, **Dr. B. Mazhari**, **Dr. A.K. Dutta** and **Dr. Ajay Jain**, whose ideas and views have greatly influenced my way of thinking.

I am also thankful to all my friends who made my stay memorable at IIT, Kanpur. I never forget the long chatting sessıons I spent with Sasi. Last, but not the least, I am indebted to my parents and my brother and sister for providing me the moral support and inspiration.

Abstract

Formal Verification based on Symbolic Model Checking has been successfully employed in the prefabrication verification of several System-On-Chip (SOC) designs. However, Symbolic Model Checking is beset with state and memory explosion problems which limits the size of designs that can be verified. SOC designs being inherently large, need to be verified using an approach based on "divide and conquer". Such an approach is, generally, taken intuitively by designers, based on the detailed knowledge of a design.

In the present thesis, we verified the Cache Coherence Protocol of a split transaction bus of Pentium Pro (P6) for a case where it is reported in the literature that the Symbolic Model Checking approach fails due to state explosion. We show how the above system can be verified using a Compositional Verification approach known as *Assume-Guarantee* approach. We illustrate how designers can leverage their detailed knowledge of a design to partition it at the module level, and thereby, enable the *Assume-Guarantee* approach to overcome intrinsic limitation of a formal verification tool to successfully verify large designs.

Contents

1 Introduction 1

1.1 Formal Hardware Verification . . . 2

1.1.1 Theorem Proving . . . 2

1.1.2 Model Checking . . . 3

1.1.2.1 Overview of Model Checking Tools . . . 4

1.1.2.2 Compositional Verification Techniques . . . 5

1.1.3 VIS: Verification Interacting with Synthesis . . . 6

1.1.3.1 Verilog front end . . . 6

1.1.3.2 Fair CTL model checking . . . 6

2 Implementation of Cache Coherence Protocol 8

2.1 Pentium-Pro Architecture . . . 10

2.1.1 Split-Transaction Bus . . . 10

2.1.2 Cache coherence in Shared-Memory Architectures . . . 11

2.1.3 Transactions on the Pentium Pro Bus . . . 13

2.1.4 IOQ . . . 16

2.1.5 MESI protocol . . . 16

2.1.6 Pentium Pro Cache Features . . . 17

2.2 Design Approach . . . 18

2.2.1 Simplification . . . 18

2.2.2 Processor . . . 19

2.2.2.1 Execution Unit . . . 21

2.2.2.2 Cache Controller . . . 21

2.2.2.3 Request/Snoop Agent 23
2.2.3 Memory 26

3 Motivation and Problem definition 35
3.1 Design Issues 35
3.2 Verification approach 37
3.2.1 Introduction to Temporal Logic and CTL 37
3.2.2 CTL properties 39
3.2.3 Verification results 40
3.3 Motivation and Problem definition 42

4 Compositional Verification techniques 44
4.1 Compositional verification 45
4.1.1 Compositional Minimization 47
4.1.1.1 Disadvantages and Advantages of the approach 47
4.1 2 Assume-Guarantee Reasoning 47
4.1.2.1 Disadvantages and Advantages of the approach 48

5 Verification approach using Assume-Guarantee technique 50
5.1 Verification Approach 51
5.1.1 First Approach: Processor0 as main machine 52
5.1.1.1 Assumptions on external inputs 52
5.2 Second approach: Request agent of Processor0 as main machine 58
5.2.1 Assumptions and Guarantees on Request agent 59
5.2.1.1 Assumptions on external signals 59
5.2.1.2 Guarantee on REQ_GRANTED 61
5.2.1.3 Transaction order (IOQ) property 62
5.2.2 Assumptions and Guarantees on combination of Execution unit and Cache module 63
5.2.2.1 Assumptions on combination of Execution unit and Cache module 63
5.2.2.2 Guarantees 65

5.2.3 Guarantees on Processor1 66
5.2.4 Guarantees on Memory 67

6 Results, Conclusion and Scope for future work 69
6.1 Results 69
6.1.1 First approach: Partitioning at the Processor level 70
6.1.2 Second approach: Partitioning at the module level of Processor0 . 70
6.1.2.1 Request Agent: 70
6.1.2.2 Combination of Execution Unit and Cache 72
6.1.2.3 Request Agent of Processor1 73
6.1.2.4 Memory 73
6.2 Conclusion 74
6.3 Scope for future work 75

Bibliography 76

Appendix 79

List of Figures

2.1 Shared-Memory Architecture 9
2.2 P6 System(simplified) Overview 20
2.3 FSM of Execution Unit 22
2.4 FSM that controls access to acquire the bus. 24
2.5 Transaction Phases 26
2.6 Block diagram of Processor 28
2.7 Block diagram of P6 bus system 29
2.8 FSM of Cache unit 30
2.9 Flow chart which represent combinational logic block in Cache unit . 31
2.10 FSM1 of Request Agent corresponding to transaction 1. 32
2.11 FSM 2 of Request Agent corresponding to transaction 2 33
2.12 FSM of Memory module corresponding to transaction 1 34

3.1 Computation tree from the state transistion graph 37

4.1 Compostional Minimization 46

5.1 FSM that constrains ADS_O_1 53
5.2 FSM that constrains inputs EXT_TRANS_REQ_O_1 55
5.3 FSM that constrains HIT_HITM_O_1 56
5.4 FSM that constrains INT_TRANS_REQ 60
5.5 FSM that constrains SNOOP_REQ 64

List of Tables

6.1 Model size in the first Approach 70
6.2 Reachabilty analysis in first approach 70
6.3 Model size of Request agent in second approach 71
6.4 Reachability analysis of request agent for case 1 71
6.5 Model size in case 2 of approach 2. 72
6.6 Reachability analysis in case 2 of second approach 72
6.7 Model size when Execution unit + Cache as main machine 72
6.8 Reachability analysis when Execution unit + Cache as main machine 73
6.9 Model size when Memory as main machine 73
6.10 Reachability analysis when Memory as main machine 74

Chapter 1

Introduction

The correctness of a hardware design is a crucial objective. Hardware designs are growing in scale and functionality. Because of this increase in complexity, the likelihood of subtle errors is much greater. Moreover, some of these errors may cause catastrophic loss of money, time or even human life. A major goal of hardware design is to enable designers to construct systems that operate reliably despite the complexity. One way of achieving this goal is by using Formal Hardware Verification: the formal proof of correctness of a design with regard to a given specification. Besides, the increasing deficiencies of traditional simulation to cope with complex designs and the breakthrough of new algorithms have drawn attention to formal verification.

In the design of digital systems, there are two different levels of verification, corresponding to two major phases of design. The first phase of design is transferring ideas into an initial description, using a high-level description language, like VHDL or Verilog. *Design verification* is used at this level to answer the question "Is what I specified what I wanted?". The second phase of design is synthesizing the initial description into a circuit which can be implemented. *Implementation verification* is used at this level to answer the question "Is what I synthesized what I specified?"; this capability is fairly developed and is embedded in many synthesis packages today.

There are two major approaches to design verification. The traditional approach is *simulation*, which is well understood and has been applied widely in the design community. The use of simulation is getting more difficult. In particular, as designs become more complex through the introduction of aggressive pipelining and concurrently-operating subsystems, it becomes increasingly difficult to anticipate the many very subtle interactions between logically unrelated system activities. For example, in a heavily pipelined machine it is often impossible to simulate all possible instruction sequences that might lead to a trap. The difficulty here is that instructions that are logically unrelated all of a sudden are being processed in parallel and may even be processed out of order. Thus, the "quality" of the validation achieved by traditional simulation is rapidly deteriorating as the VLSI technology progresses.

The second approach is *formal design verification,* which is the process of mathematically proving that a system possesses a given property. Formal hardware Verification [1,2] attempts to overcome the weakness of non-exhaustive simulation by proving the correspondence between some abstract specification and the design at hand. In the long run these formal approaches to hardware validation are better able to scale with the complexity of VLSI designs. The main reasons for this is that they exploit powerful tools of mathematics rather than brute-force. Formal hardware verification has recently attracted considerable interest. The need for "correct designs", coupled with the major cost associated with products delivered late, are two of the main factors behind this.

1.1 Formal Hardware Verification

Two well established approaches to formal verification are Model Checking and Theorem proving. In the later sections, we will be giving a very brief description of Theorem proving and a detailed description of Model Checking.

1.1.1 Theorem Proving

This is one of the earliest approaches to formal hardware verification. Theorem proving [2,3] is a technique where both the system and its desired properties are expressed as formulas in some mathematical logic. This logic is given by a formal system, which defines a set of axioms and a set of rules. Theorem proving is the process of finding a proof of a property from the *axioms* of the system. Steps in the proof appeal to the axioms and rules, and *possibly* derived *definitions* and intermediate *lemmas*. While proofs can be constructed by hand, it is very tedious. The main advantage of a formal proof is that it can be mechanically checked. The main disadvantage is that it requires a large amount of effort on the part of the user in developing specifications of each component and in guiding the theorem prover through all of the lemmas.

1.1.2 Model Checking

One of the characteristics of the Theorem proving approach is its structural rather than behavioural view of the verification process. Model Checking [4 - 8] takes the latter approach. Here, only the behaviour of some system is checked and verified to satisfy some properties. In general, model checking is an algorithm that can be used to determine the validity of formulas written in some temporal logic with respect to a behavioural model of a system. For example, it can be used in verifying communication protocols to prove properties such as an interrupt is acknowledged at most *t* time units after the interrupt request.

Temporal Model Checking [6,7] was invented in 1981 by Clarke and Emerson. This technique made it possible to algorithmically verify logical properties, expressed in a notation called Temporal Logic, of finite-state systems. The main strength of these temporal logics is the fact that the decision procedure is completely automated. Thus, the user never needs to be aware of the CTL structure, but can simply interact with the model checker to determine that the dynamic behaviour of the design satisfies the specification.

There are *two* major drawbacks with temporal logics and model checking. *First*, in this approach to Hardware Verification, it is often quite difficult to judge whether the temporal formulas that has been checked, completely characterizes the desired behaviour of the system. Also, for a practical point of view, the temporal logic formulas can sometimes become exceedingly difficult to understand. Hence, there is a danger of misunderstanding what properties actually have been verified. The *second* major drawback with temporal logic is related to its decision procedure. In order to determine the validity of a temporal logic formula it is necessary to extract the CTL structure for the device. If there are many finite state processes in parallel which are interacting, then the total state space can be enormous This problem is usually referred to as the "*state explosion problem*" The topic of how to deal with this problem is currently a very active area of research The most promising approach is symbolic methods [8] In 1987, McMillan used Bryant's *ordered binary decision diagrams* to represent state transitions efficiently, thereby increasing the size of the systems that could be verified In this approach the explicit construction of the state graph is avoided Instead, the state graph is represented by means of functions from sets of states to sets of states In 1987, McMillan developed a software tool called SMV (Symbolic Model Verifier) with which systems with over 10^{20} states could be verified

Although symbolic model checking has significantly increased the state space that can be verified, it is still too small to model circuits that include non-trivial data paths Hence, model checking is primarily used to verify the critical parts of a design, e g , cache coherence protocols, or a complex pipeline scheduling scheme, like out-of-order execution

1.1.2.1 Overview of Model Checking Tools

Numerous Model Checking tools were developed in the past decade both in academic institutions and in commercial organizations SMV (Symbolic Model Verifier), is the first Symbolic Model Checker developed by McMillan in CMU[1] COSPAN, a system developed in Bell labs, offers *language containment* University of California, Berkeley developed HSIS (Hierarchical Sequential Interactive System) University of Colorado and University of California, Berkeley are working on VIS [9] They released a recent version of VIS, VIS-1.3. VIS combines the model checking and language containment features Extensive user documentation is available for VIS on the World Wide Web [9]

Many commercial tools are available Cadence's Model Checking tool 'FormalCheck' is very popular In Fujitsu, a state-of-art Symbolic Model Checking tool BINGO is used The model checker SVE developed at Siemens is applied to various internal development projects Many companies made Symbolic Model Checking as a part of design cycle Several companies have integrated symbolic model checking into proprietary design

[1]Carnegie Mellon University

tools For example, the RuleBase system developed at the IBM Research lab integrates in it a Symbolic Model Checker

Now-a-days, Symbolic Model Checking tools are routinely expected to handle designs with 10^{120} reachable states

1.1.2.2 Compositional Verification Techniques

Many composing finite state processes in parallel leads to the state explosion problem, which is mentioned in section 1 1 2 This imposes a strong limit on the size and complexity of a system that can be verified by state enumeration methods like symbolic model checking Since most systems by design have a natural division into components or modules, it seems natural to try to reduce the number of states by taking a divide-and-conquer approach This means that we separate the specification of system into properties on its components These kind of techniques, in literature, are called 'Compositional Verification' techniques [10-14] Assume-Guarantee approach is a Compositional Verification technique Roy et al [15,16] have successfully verified a class of properties on AOF (audio output interface) [16] and PicoJava EBI [15] modules by employing the Assume-Guarantee approach [10,16] using state-of-art symbolic model checking tool, BINGO It is important to note that at present no commercial symbolic model checking tool supports Compositional Verification as a front end More about the Assume-Guarantee approach is discussed in Chapters 4, 5. In our present work, we employ Assume-Guarantee approach to verify the cache coherence protocol of a split transaction Pentium Pro (P6) bus, where it is reported in [1] that symbolic model checking failed due to the state explosion problem

The chapters are organized as follows Chapter 2 discusses about cache coherence protocol and its implementation [1] in Verilog [17] Chapter 3 discusses about the results given in [1] Chapter 4 gives a brief introduction on Assume-Guarantee approach Chapter 5 discuss about the verification approach taken for cache coherence protocol [18,19] of the P6 bus using an Assume-Guarantee approach Chapter 6 discusses some results and scope for future work

1.1.3 VIS: Verification Interacting with Synthesis

In our present work, we verify the cache coherence protocol using the verification tool, VIS A brief introduction of the capabilities and features of VIS is discussed below Extensive user documentation of VIS is available on the World Wide Web [9]

VIS is a tool that integrates the verification, simulation, and synthesis of finite-state hardware systems It uses a Verilog front end and supports fair CTL model checking, language emptiness checking, cycle-based simulation and hierarchical synthesis

VIS is divided into three parts a common front end for reading in a description of a design, verification and Synthesis We briefly discuss about the first two

1.1.3.1 Verilog front end

VIS operates on an intermediate format called BLIF_MV, which is an extension of BLIF, the intermediate format for logic synthesis accepted by SIS [20] VIS includes a stand-alone compiler from Verilog to BLIF_MV, called *vl2mv*, which supports a synthesizable subset of Verilog *vl2mv* is built to connect between various existing designs in Verilog and powerful synthesis/verification algorithms in HSIS/SIS *vl2mv* extracts a set of finite state machines (FSMs) which preserve the behaviour of the source Verilog programs which is defined in terms of simulated results When a BLIF-MV description is read into VIS, it is stored hierarchically as a tree of modules which in turn consists of sub-modules Verification operations can be performed at any subtree of the hierarchy

1.1.3.2 Fair CTL model checking

Computation Tree Logic (CTL) model checking is a technique pioneered by Clarke and Emerson to verify whether a finite state system satisfies properties expressed as formulas in a branching-time temporal logic called CTL More about CTL [9] is discussed in Chapter 3 VIS performs fair CTL model checking under Buchi fairness constraints [9] Buchi fairness constraints are those states that are visited infinitely often The language of a design is given by sequences over the set of reachable states that do not violate the fairness constraint If model checking fails, VIS reports the failure with a counterexample, (i e , behaviour seen in the system that does not satisfy the property) This is called the "*debug*" trace Debug traces lıst a set of states that are on a path to faır cycle and faıl the CTL formula

Chapter 2

Implementation of Cache Coherence Protocol

With the ever increasing need for computing power, multiprocessor systems with cache-coherent shared-memory architecture, as shown in Fig 2 1, have gained prominence One of the primary challenges in these systems is the synchronization and control of shared information To maintain integrity of all memory accesses, whenever one processor updates a memory block, the change must be propagated to all the processors that have cached the same memory block Many protocols have been proposed and implemented, but only a few have been formally verified Formal verification of the cache coherence protocols is beset with problems such as, the state explosion problem [see section 1 1 2] and large computation time Random simulations, on the other hand, are not very reliable, as the random test sequence cannot explore all the reachable states

Chapter 2 and Chapter 3 discusses the work reported in [1] In this, a split-transaction bus of Pentium Pro (P6) is modeled in Verilog and VIS is used to verify the cache coherence protocol employed in P6 P6 bus is tuned for multi-processor environment with pipelining Data cache of P6 uses the MESI cache coherence protocol to ensure cache consistency among the different processors Because of bus pipelining the state space generated is enormous. To overcome the possibility of encountering the state explosion problem due to enormous state space, the model is simplified without compromising the problem complexity In [1], the simplification steps involved reduction of memory size,

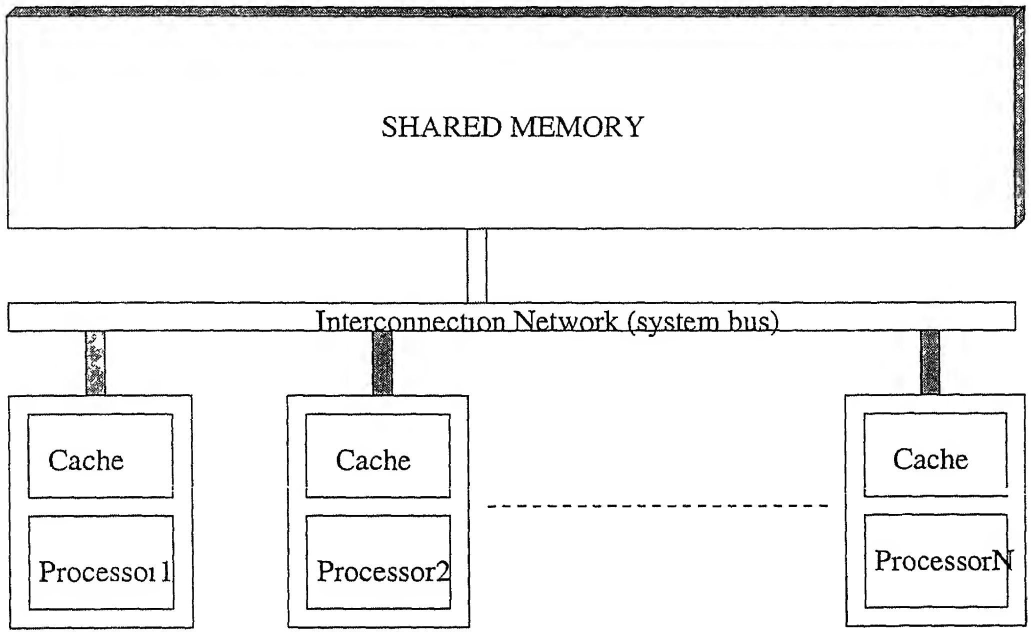


Figure 2 1 **Shared-Memory Architecture**

cache size, data width, number of processors, and maximum number of transactions that the bus can handle We use the same model

In this chapter, the simplified implementation model of the cache coherence protocol is discussed Much of the description that follows is taken from [1] However, we attempt to explain the implementation model with more clarity by basing our discussion on state transition graphs of the various modules employed and were extracted from the Verilog code The chapter is organized as follows Section 1 outlines the features of Pentium-Pro relevant to the present work Section 2 discusses the design implementation with respect to the state transition graphs

2.1 Pentium-Pro Architecture

2.1.1 Split-Transaction Bus

A split-transaction bus is a pipelined bus in which bus transaction is split into two subtransaction, typically, *request* and *response* transaction that arbitrate for the bus separately, so that multiple transactions can be outstanding at a time on the bus. After an agent (typically processors, I/O controller, memory controller) gains control of the bus, it initiates a transaction. A typical transaction consists of a request and a data response. The request consists of a transaction type (e.g. read line, read and invalidate line, write back), an address and identity of the transaction. When data is returned, the transaction ID is driven with the data. Other transactions are allowed to intervene between them, so that the bus can be used while the response to the original request is being generated. Buffering is used between the bus and the cache controllers to allow multiple transactions to be outstanding on the bus waiting for snoop and/or data responses from the controllers.

The advantage, of course, is that the bus is utilized more effectively by pipelining bus operations and more processors can share the bus. The disadvantage is the increased complexity.

The major complications caused by the split-transaction are [18,19]:

1. A new request can appear on the bus before the snoop and/or servicing of an earlier request completes. In particular, *conflicting requests* (two requests to the same memory block, at least one of which is due to a write operation) may be outstanding on the bus at the same time. Avoiding conflicting requests is easy [19]:
   A simplest approach is to use the broadcast capability of the bus. All coherence controllers track every bus accesses. Every controller can keep track of the memory address of any outstanding bus requests, since it can see the request and the corresponding reply on the bus. When the local processor generates a miss, the controller does not place the miss request on the bus until there are no outstanding requests for the same cache block.
   Alternatively, we could have each processor buffer only its own requests and track

the responses of other processors. If the address of the requested block were included in the reply, then the second processor to request the block could ignore the reply and reissue its request.

2. The number of buffers for incoming requests and potential data responses from the bus to cache controller is usually fixed and small. For Pentium Pro the buffer size is eight. So, the flow of the transactions have to be handled carefully to prevent overflowing of the buffer. This is called *flow control.*

3. Request should eventually be followed by response. In Pentium Pro the approach is *in-order* . In other words, responses are sent in the order of the requests made. Out-of-order approach is more complex to handle, but is more tolerant of variations in memory access times.

2.1.2 Cache coherence in Shared-Memory Architectures

The SMP (Symmetric Multiprocessor) form of parallel architecture in Pentium Pro provides a global physical address space and symmetric access to all of main memory from any processor. Every processor has its own cache and memory modules attached to a shared-bus without any glue-logic. The architecture supports the caching of both shared and private data. Private data is used by a single processor, while shared data is used by multiple processors, essentially providing communication among the processors through reads and writes of the shared data. When a private item is cached, its location is migrated to the cache, reducing the average access time as well as the memory bandwidth required. When shared data are cached, the shared value may be replicated in multiple caches. In addition to the reduction in access time and required memory bandwidth, this replication also provides for a reduction in contention that may exist for shared data items that are being read by multiple processors simultaneously. Caching of shared data, however, introduces a *Cache Coherence Problem* i.e when two processes see the shared memory through different caches, there is a danger that one may see the new value in its cache while the other still sees the old value. This cache coherence problem needs to be addressed as basic hardware design issue; for example, stale cached copies of a shared location must be eliminated when the location is modified, by either invalidating them, or updating them with the new value.

More formally, a multiprocessor system is coherent if the results of any execution of a program are such that, for each memory location, it is possible to construct a hypothetical serial order of all operations to the location (i.e., put all reads/writes issued by all processors into a total order) which is consistent with the results of the execution and in which:

1. operations issued by any particular processor occur in the above sequence in the order in which they were issued to the memory system by that processor, and
2. the value returned by each read operation is the value written by the last write to that location in the above sequence.

Implicit in the definition of coherence is the property that all writes to a location (from the same or different processes) are seen in the same order by all the processes. This property is called *write serialization* . It means that if read operations by processor P1 to a location see the value produced by write w1 (from P2, say) before the value produced by write w2 (from P3,say), then reads by another processor P4 (or P2 or P3) should also not be able to see w2 before w1. There is no need for an analogous concept of read serialization, since the effects of reads are not visible to any processor but the one issuing the read.

Coherence is maintained by having each cache controller 'snoop' on the bus and monitor the transactions. In general, a snooping-based cache coherence scheme ensures that:

1. all "necessary" transaction appear on the bus, and
2. each cache "snoops" (monitors) the bus for relevant transactions and takes suitable actions which could be

- invalidating or updating the contents or state of that memory block and/or
- supply the latest value for that memory block from the cache to the bus.

A snoopy cache coherence protocol ties together two basic facets of computer architecture: bus transactions and the state transition diagram associated with a cache block .

2.1.3 Transactions on the Pentium Pro Bus

Each transaction initiated on the Pentium Pro bus proceeds through the following phases from its inception to its completion [20].

1. Arbitration
2. Request
3. Error
4. Snoop
5. Response
6. Data

Each transaction is associated with a signal group. Each signal group is only used during a particular phase of the transaction. At a given instance in time, maximum of eight transactions can be active on the bus at various phases on their way to completion. In those cases where the current owner(s) of a signal group is not yet done using it, the next owner(s) cannot start using it until the previous agent's done and the signal group has been completely released. Phases proceed in pre-defined order.

The following sections provide a brief description of the phases.

Arbitration Phase

The Pentium Pro processors has built-in rotation priority scheme. No external arbitration logic is necessary to decide which of them requires the request signal group and which gets it next. Each processor keeps track of the following:

- who owns the request signal group
- who owned the request signal group last
- who gets the request signal group the next

Arbitration algorithm ensures fairness in granting the bus ownership. Each processor is assigned a unique ID. Then, the *Arbitration algorithm* works as follows:

If there are $n \in N$ processors, and if the requester has ID = m (say) ($m \in [0,n-1]$), and the last owner has ID = k(say) ($k \in [0,n-1]$), then this requester (with ID = m) gets the request if none of the processors with IDs (k+1)%n, (k+2)%n, ..., (m-1)%n is requesting for bus ownership.

Request Phase

The request agent uses the request signal group to issue the transaction request to other bus agents. These bus agents latch the request. The snoop agents determine if it's a memory transaction. If it is, they must perform a cache lookup and deliver the snoop result during the snoop phase of this transaction.

When a request agent has finished driving it transaction request onto the the request signal group, another request agent can take ownership of the request signal group and issue another transaction.

Error Phase

The other bus agents check the parity of the request just latched. If the parity is correct, no action is taken. If one or more of them detect a parity error, they each assert AERR signal at the end of the error phase. The request agent checks the AERR signal at the end of the error phase to see if they all received the request without error. If there was an error, the remaining phases of the transaction are canceled.

Snoop Phase

Request agent then proceeds to the snoop phase (if there was no error) and begin sampling the snoop result signal group. The snoop agents are responsible for delivering the result of their cache snoop during this phase. Once the snoop result has been received, the request agent stops sampling the snoop result signals and the snoop agents cease driving them.

Note that snoop phase can be stretched by the snoopers if any of them needs a little time in order to produce the results of the cache lookup.

Response Phase

Having completed the snoop phase, the request agent proceeds to the response phase and begins sampling the response signal group. The response agent is responsible for delivering following responses:

- tells the request agent to retry transaction later.
- indicates a hardware failure
- if the request is a write, it will accept the data in data phase.
- if a request is a read and snooper did not indicate a hit on a modified line, it will supply the data during the data phase.
- if a snooper indicated a hit on modified line, the entire line will transferred from the snooper to memory (and, if its a read, to the request agent as well)–referred to as an implicit write-back response.
- instructs the request agent to end the transaction with no data transferred.

Once the response has been received, the request agent stops sampling the response signal group and the response agent ceases to drive the response.

Note that this phase can be stretched by the currently-addressed target if it requires extra time before it decides what its response to the transaction will be.

Data Phase

If the transaction involves a data transfer, the request agent proceeds to the data phase of the transaction. If it's a write transaction, the request agent takes ownership of the data bus and delivers the data to the response agent during this phase. If it's read transaction, the response agent takes the ownership of the data bus and delivers the data to the request agent. Once the data phase completes, the transaction has completed.

2.1.4 IOQ

In order to properly interact with the bus at each stage of the appropriate transactions, each bus agent must maintain a record of all transactions currently in progress, what phase each is currently in, and what responsibilities (if any) it has during each phase. This record is kept in a buffer referred to as the agent's *in-order queue*, or *IOQ* . When a transaction receives a response guaranteeing that the transaction will be completed now, the transaction is deleted from the queue.

2.1.5 MESI protocol

One of the popular cache coherence protocol implemented in hardware is the MESI [18]. Each cache line includes state bits. The values taken by these state bits depend on what the cache controller does during its bus masters's data transfer activities, and also during the snooping activities it performs in response to snoop requests generated by other bus masters.

M.E.S.I states

Cache line can be in any one of the following states.

Modified [M]: This state indicates a line which is available only in this cache, and no other cache line or Memory has the latest data. Such a modified line can be updated locally in the cache without acquiring the bus.

Exclusive [E]: Indicates a line which is available only in this cache and Memory. No other cache line has a valid copy of this line.Writing to an exclusive line causes it to change to the Modified state and can be done without informing other caches, so no bus activity is generated.

Shared [S]: Indicates that this line is potentially shared with other caches (the same line may exist in one or more of the remaining caches). A shared line may be read by the CPU without a main memory access. Writing to a shared line updates the cache and also requires the cache controller to generate a write-through cycle to the bus. The write-through cycle will invalidate this line in other caches.

Invalid [I]: Indicates that this line is not available in the cache. A read to this cache line will be a miss and cause the cache controller to execute a line fill (fetch the entire line and deposit it into the cache SRAM). A write to this cache line will cause the cache controller to execute a write-through cycle to the bus. After the line fill, the state of the line changes to the exclusive state since Memory also has a valid copy.

Bus state transistions: The states determine the the actions performed by the cache controller with regard to any activity related to a line, and the state of a line may change due to these actions. A snoop hit for a read on the Modified line, or Exclusive line, changes it to a Shared state. A snoop hit for a write, changes the state of the line to an Invalid state.

2.1.6 Pentium Pro Cache Features

Pentium Pro is tuned for multi-processing. The bus can handle up to eight transaction. It has two levels cache [4]:

1. Level I Cache has two separate Data and Code cache.

   Data It is 8KB with 2-way set-associative. It uses MESI protocol to maintain the data coherency.

Code It is 8KB with 4-way set-associative. It uses a subset of SI of MESI protocol.

2. Level II Cache is unified data and code cache. It is 256KB with 4-way set-associative. It uses MESI protocol. It backs-up the L1 data and code cache.

2.2 Design Approach

2.2.1 Simplification

Modeling the split-transaction Pentium Pro bus with all the transaction (pipelining) queues and actual data and address widths supported by Pentium-pro results in enormous state space and the verification is not possible due to state-explosion problem. Hence, in [1], the design is simplified as given below (Fig. 2.2).

1. The P6 System has following blocks:
   - Two Processors
   - One response Agent i.e. memory
2. Bus can support only 2 transactions. Transaction phases are restricted to:
   - Arbitration
   - Request
   - Snoop
   - Response (Data phase is merged)
3. Each processor has the following blocks:
   - Execution Unit
   - Cache Controller with cache
   - Request/Snoop block

4. Data path is 1 bit. Memory has been organized into 4 X 1 bit.

5. Cache size is 2 lines X 1 bit and is direct-mapped. Only L1 data cache is implemented. Moreover, the L1 data cache are Write-Back memory type. There is a data TLB which supports the direct-mapping.

6. Instruction LOAD, STORE, LOAD2STORE, and INC is supported. LOAD2STORE instruction simulates the speculative read feature of Pentium Pro.

   Only memory transaction have been incorporated. The following notation is used to indicate different transaction types.

For Requests From the Execution Unit to Cache:

MR	Read (from Cache)
MW	Write (to Cache)
MRI	Read (from Cache) and Invalidate (other Cache)

For Requests From the Cache to the bus:

EXT_MR	Read (from other Cache/Memory)
EXT_MW	Write (to Memory)
EXT_MRI	Read (from other Cache/Memory) and Invalidate (other Cache)
EXT_MRI_Z	Invalidate (other Cache)

2.2.2 Processor

Each processor has the following blocks:

1. Execution Unit

2. Cache Controller with cache

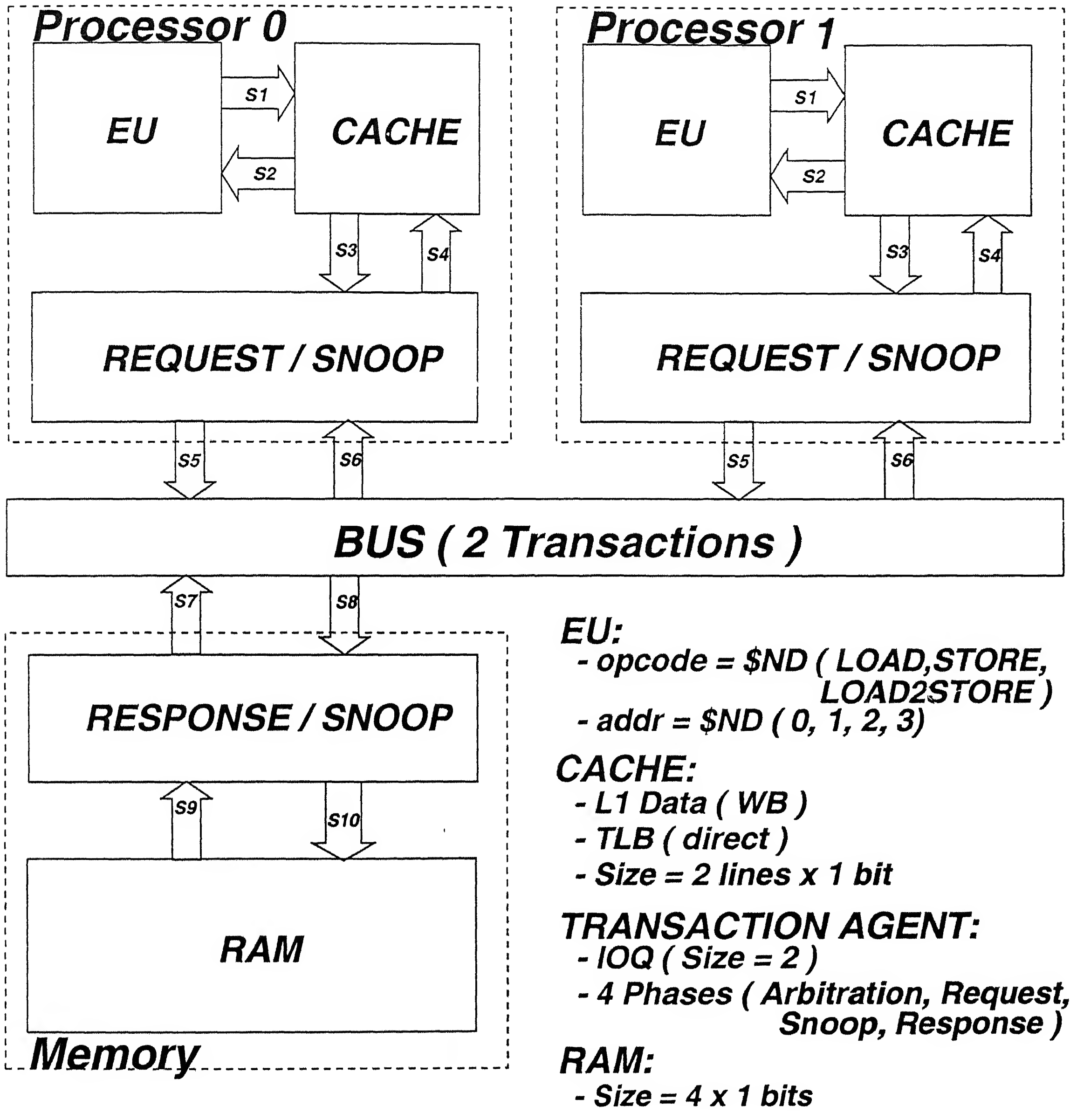


Figure 2.2: **P6 System(simplified) Overview**

3. Request agent/Snoop block

   Each block is explained with respect to state transition graphs that are extracted from the Verilog code.

2.2.2.1 Execution Unit

Execution takes place in 3 stages :

FETCH Fetches instructions randomly from { LOAD, STORE, LOAD2STORE, INC } and address from the { 0, 1, 2, 3 }. Goes to EXECUTE stage.

EXECUTE If instruction is not INC, it goes to next stage i.e., MEM stage else goes back to FETCH.

MEM Puts the request to the cache controller (CACHE_REQ) for the desired action. It goes back to the next FETCH stage once the cache controller gives the signal, CACHE_RDY or DATA_RDY (Fig. 2.3).
The state transistion graph for the Execution unit is shown in Fig. 2.3.

2.2.2.2 Cache Controller

Cache controller accepts request from the execution unit as well as snoop request from the snooper. Based on the snoop request from the snooper and requests from the execution unit, the cache controller implements the MESI cache coherence protocol.

The state transition graph (STG) for the FSM of cache controller which implements the MESI cache coherence protocol is given in Fig. 2.8. The signals used in the state transition graph are with respect to Verilog code. The variable names employed are readable and self-explanatory. As shown in Fig. 2.8, the edges drawn with continuous lines corresponds to transitions occurring due to requests from the Execution Unit (EU). For example, let the initial state of the cache line be **S1**. If a read miss occurs for a requested cache line by EU, the cache controller, in the first instance, has to replace an

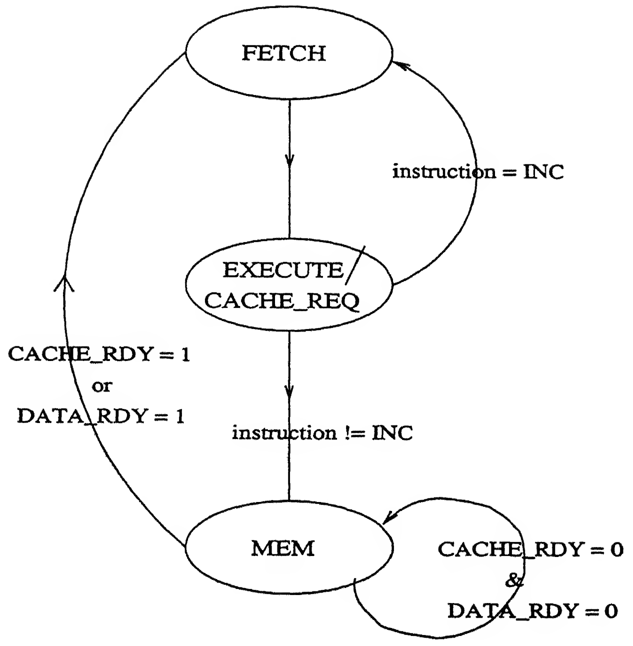


Figure 2.3: **FSM of Execution Unit**

existing cache line depending on the replacement algorithm used. If the cache line to be replaced is in a Modified state (*writebackrequired* = 1, in the code), a 'write-back' cycle is initiated by the cache controller wherein the Modified line is written back to Memory. With respect to the STG, it goes from the initial state **S1** to **S11'** and when the bus request is granted, it write-backs the Modified data to Memory and goes back to **S1**. Then, the cache controller executes a *line fill* (fetch the entire line from another processor or main Memory and deposit it into the present cache). With respect to the STG, it goes from **S1** to **S12** and when the bus is granted, the cache line is fetched and the state of the line changes from Invalid to either Exclusive or Shared depending in whether the line is fetched from another processor or Memory. This can be observed in the Fig. 2.8.

The lines in the state transition graph marked with dotted lines correspond to transitions that occur due to the activities on the bus. For example, consider the state of the cache line to be in **S3** (Fig. 2.8), i.e., in Modified state. If another processor floats a transaction on the bus requesting for the cache line (say, *EXT_MR*), the state of the cache line changes from Modified to Shared (**S3** to **S2**) and the requested cache line is delivered onto the bus. The transitions on dotted lines is due to the combinational logic in the cache module (see Fig. 2.8). The combinational logic is shown as a flow-chart in Fig. 2.9. It accepts snoop requests through the signals, SNOOP_REQ and SNOOP_TAG_I (see Fig. 2.6) of the request agent and changes the cache state bits based on the snoop request.

In the conflicting scenario, where a cache controller is busy updating the cache memory, and at the same time a snoop result for the same line is requested, the snoop result is delayed and a stall is generated. The cache line is updated accordingly. This is seen in Fig. 2.9, where *SNOOP_TAG_O = STALL* is generated, when the cache controller is updating the same cache line that is requested by another processor, i.e., when the condition *SNOOP_ADDR = ADDR* is satisfied.

2.2.2.3 Request/Snoop Agent

This agent requests, as well as, snoops the bus transactions that take place on the bus. It maintains the transaction record in a FIFO with size two (as there can be a maximum of two transactions in the simplified case).

As a requester

On getting the request from the Cache Controller, i.e., *INT_TRANS_REQ != NOREQ* (Fig. 2.6), it goes into the Arbitration Phase. It asserts the signal BR0 and goes to state **SB** from the initial state **SA**. This is as shown in the above Fig. 2.4. In this state, i.e., **SB**, it checks if it has the right to take over the bus.
It checks for the following conditions.

1. If both the processors are requesting the bus at the same time, then according to the Arbitration algorithm (as discussed in section 2.1.3) only one processor gets access to the bus.

2. If an other processor acquires the bus, then it has to wait until the next clock cycle.

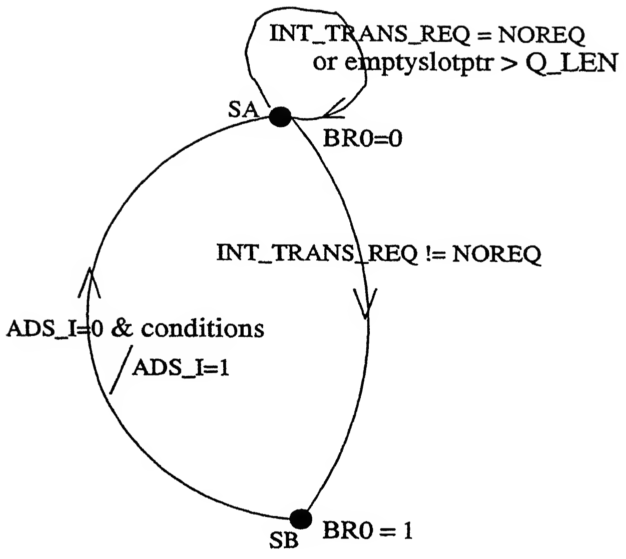


Figure 2.4: **FSM that controls access to acquire the bus.**

3. If another processor is trying to write into the same cache line that is requested by the Request agent (of the local processor), then the Request agent (of the local processor) cannot place its request on the bus until all outstanding requests for the same cache line have been serviced. This is because, if the Request agent (of the local processor) is allowed to acquire the bus and place its request, then the Memory will respond to both requests (as the bus has split transaction capability) and it results in two caches having the same cache line in Modified state, which is inconsistent with MESI protocol.

If all the above conditions are satisfied, then it asserts ADS_I and puts the following information on the bus, *Request Type, Address, ID*. If any of these conditions are not satisfied, then it keeps the BR0 line asserted until it gets the access to the bus.

As a snooper

When the ADS line is asserted, it enqueues the transaction in the FIFO ioq's.

The transaction goes from IDLE phase to REQUEST phase (see Fig. 2.10 and

Fig.2.11). If the transaction ID is not the same as the snooper ID, then it sends a request (SNOOP_REQ, see Fig. 2.6) to cache controller to get the state of the cache line.

The transaction goes into SNOOP Phase. If the transaction ID is not the same as the snooper ID, then it waits for SNOOP_TAG_I from Cache unit. As discussed in section 2.2.2.2, if the cache controller is busy, then it makes *SNOOP_TAG_I = STALL* (Fig. 2.9) making the transaction to stall in **S3**. When the transaction ID is same as the local processor ID, then it moves from **S3** to **S3a** where it waits for HIT_HITM_I from another processor. If the cache controller of another processor is busy then it makes *HIT_HITM_I = STALL* causing the transaction to stall in **S3a**. Once there is a valid snoop result, the transaction moves from SNOOP phase to RESPONSE phase (**S3a** to **S4**).

In the response phase, if the cache state is modified, then it also gets a copy of the data which it supplies later in the response phase (When it sees the TRDY signal is high). Once the request is granted (*REQ_GRANTED =1*, see Fig. 2.6), the snooper indicates this to cache Controller (if the ID of the transaction matches that of the snooper) and sends (if required) the data to cache.

There are two FSM's in the Request Agent module corresponding to two transactions that are outstanding at a time on the bus. As seen in 2.11, when the second transaction is in a particular phase, it cannot move to the next phase until the first transaction releases the signals corresponding to that phase. On getting the TRAN_OVER signal from the Memory Controller, the Request agent dequeues the transaction from the FIFO buffer. If the second transaction is active in the buffer (in a particular phase) and if the first transaction is completed (*TRAN_OVER = 1*) then the first transaction is dequeued from the FIFO buffer and the second transaction is pushed up in the buffer. This results in the second transaction to move from RESPONSE PHASE to IDLE phase and the first transaction to move from RESPONSE phase to the phase where the second transaction is previously in. This is shown by transitions *a, b, c* in Fig. 2.10.

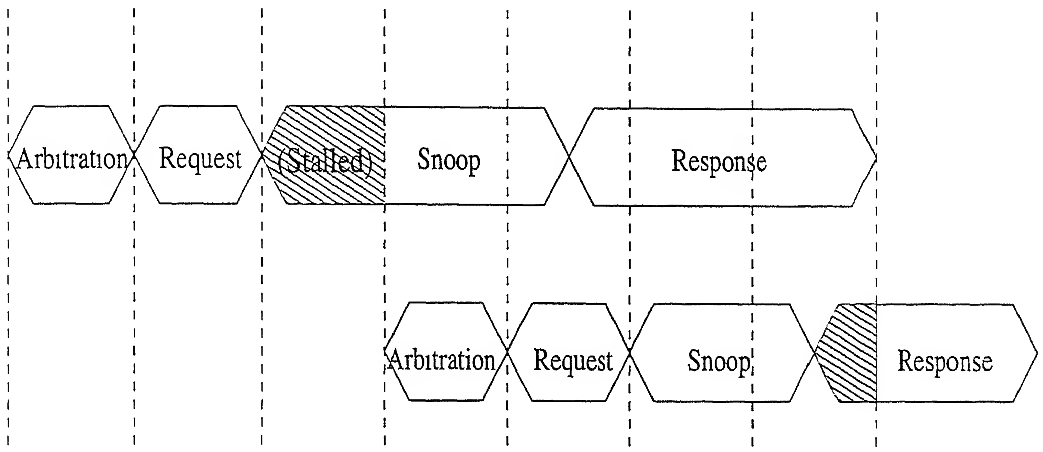


Figure 2.5: **Transaction Phases**

2.2.3 Memory

When ADS line is asserted, it enqueues the transaction in the FIFO ioq's It updates its IOQ to indicate that the transaction is in request Phase. As the memory is not required to snoop on any cache line, it just waits for one clock (Fig 2.12). And then it updates the transaction to snoop phase after one clock. In the snoop phase it checks the signal HIT_HITM_I. If *HIT_HITM_I != STALL*, it updates the transaction into response phase and does the following.

- **Request type is EXT_MR or EXT_MRI and HIT_HITM_I != MODIFIED**
  It asserts DRDY (only if *DRDY =0*). It sends the data and sets Transaction_Over signal in the next clock.

- **Request type is EXT_MR or EXT_MRI and HIT_HITM = MODIFIED**
  It asserts TRDY (if *DRDY=0*). In the next clock, the snooper with modified line seeing TRDY going high, supplies the data. In the next clock, Memory Controller grabs the data and sets Transaction_Over signal.

- **Request type is EXT_MW**
  It asserts TRDY (if *DRDY =0*). In the next clock, the requester seeing TRDY going high, supplies the data. In the next clock, Memory Controller grabs the data and

sets Transaction_Over signal.

- **Request type is EXT_MRI_Z**
  It sets Transaction_Over signal.

The FSM's corresponding to first transaction in Memory module is shown in Fig 2.12. The FSM corresponding to another transaction is similar to Fig. 2.12. Memory Controller dequeues the transaction after it sets the Transaction_Over signal (*TRAN_OVER* = *1*).

Figure 2.6: **Block diagram of Processor**

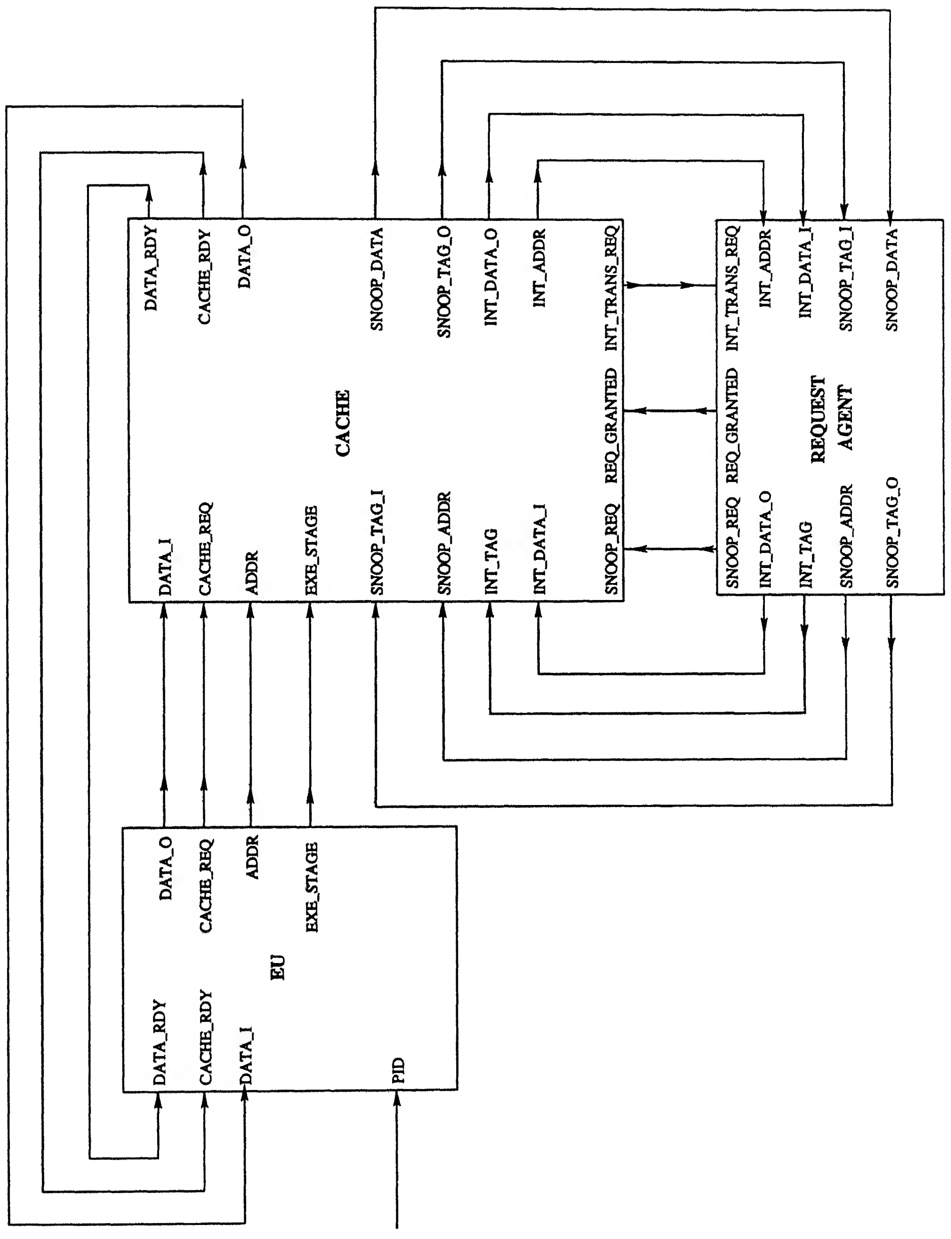


Figure 2.7: **Block diagram of P6 bus system**

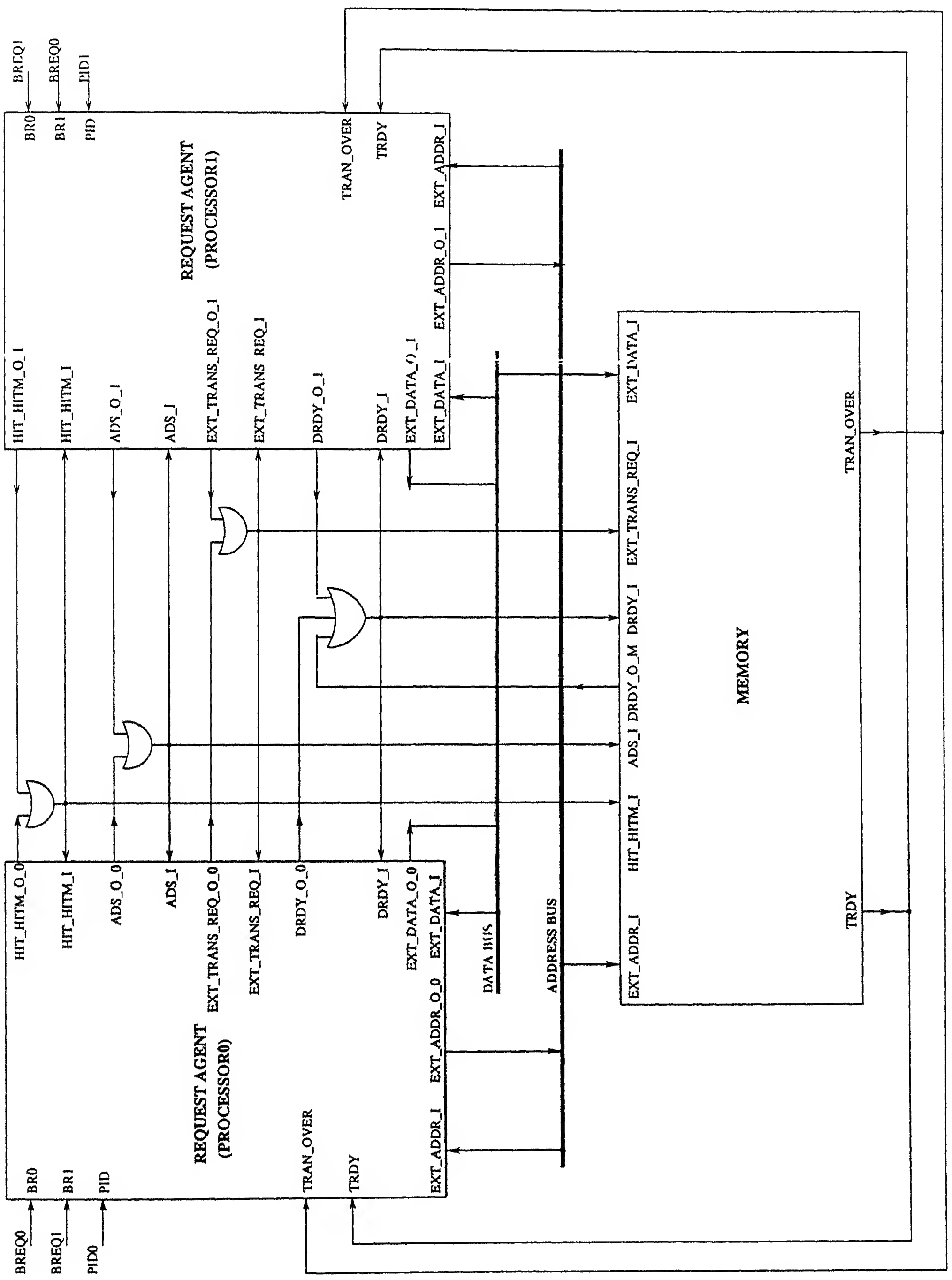
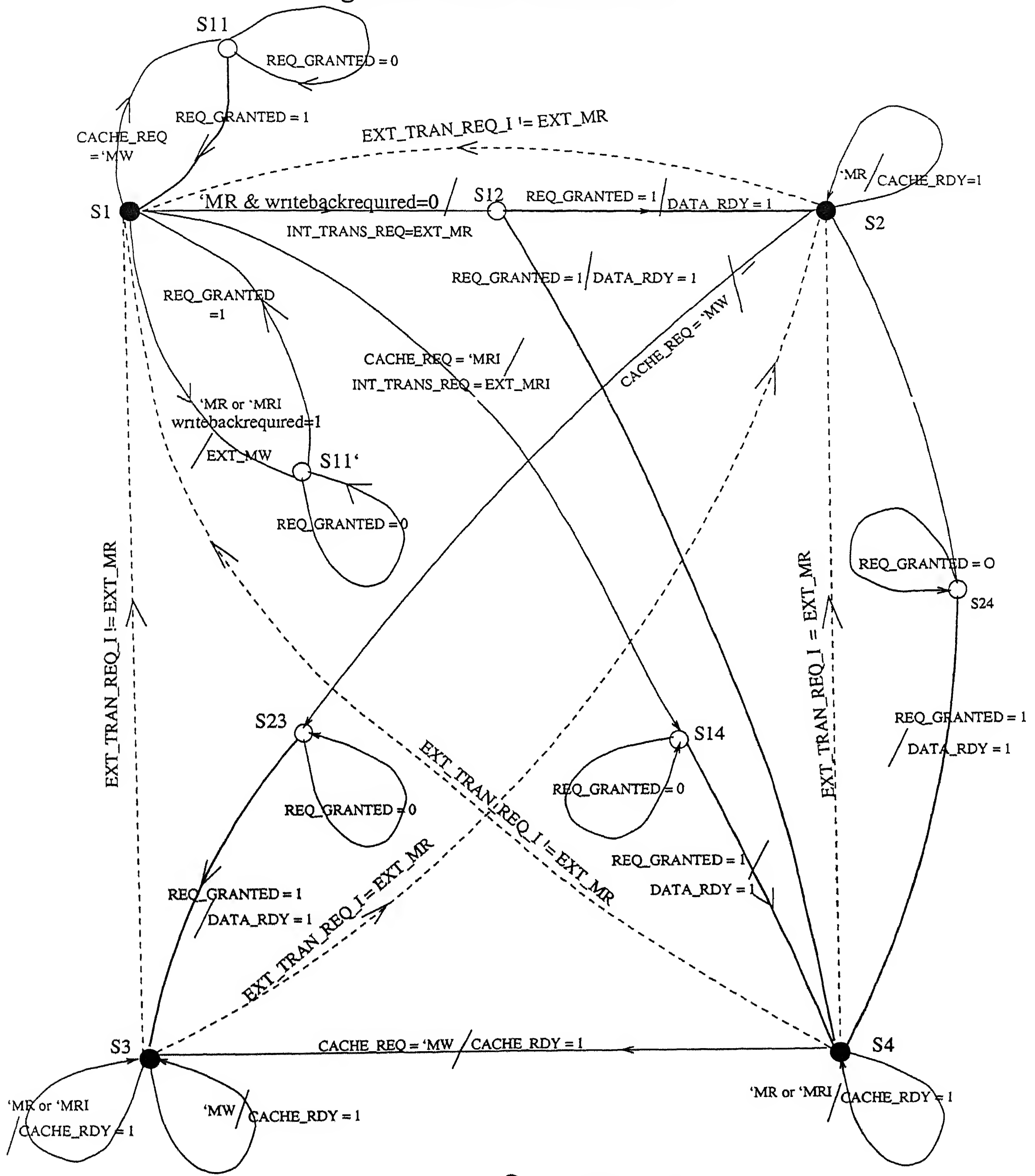


Figure 2.8: **FSM of Cache unit**



● - main states
S1 - lstate = INVALID, INT_TRANS_REQ=0
S2 - lstate = SHARED, INT_TRANS_REQ=0.
S3 - lstate = MODIFIED, INT_TRANS_REQ=0
S4 - lstate = EXCLUSIVE, INT_TRANSREQ=0
S14 - lstate = INVALID, INT_TRANS_REQ=EXT_MRI

○ - Intermediate states
S11 - lstate = INVALID, INT_TRANS_REQ=EXT_MW
S11' - lstate = INVALID, INT_TRANS_REQ=EXT_MR or EXT_MRI
S12- lstate = INVALID,INT-TRANS_REQ=MR, writebackrequired=1
S24 - lstate = SHARED, INT_TRANS_REQ=EXT-MRI
S23 - lstate = SHARED, INT_TRANS_REQ=EXT_MRI_Z

Figure 2.9: **Flow chart which represent combinational logic block in Cache unit**

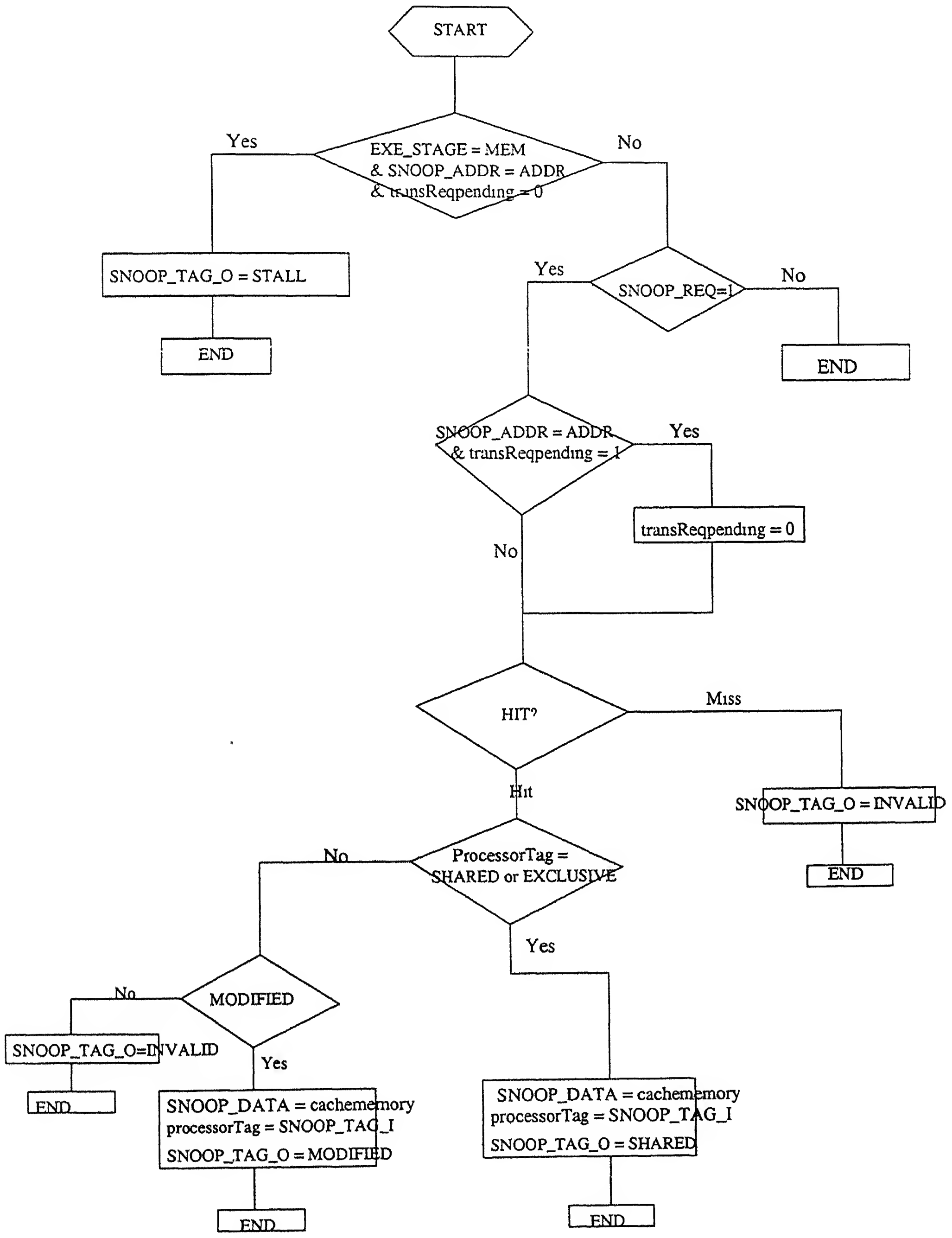
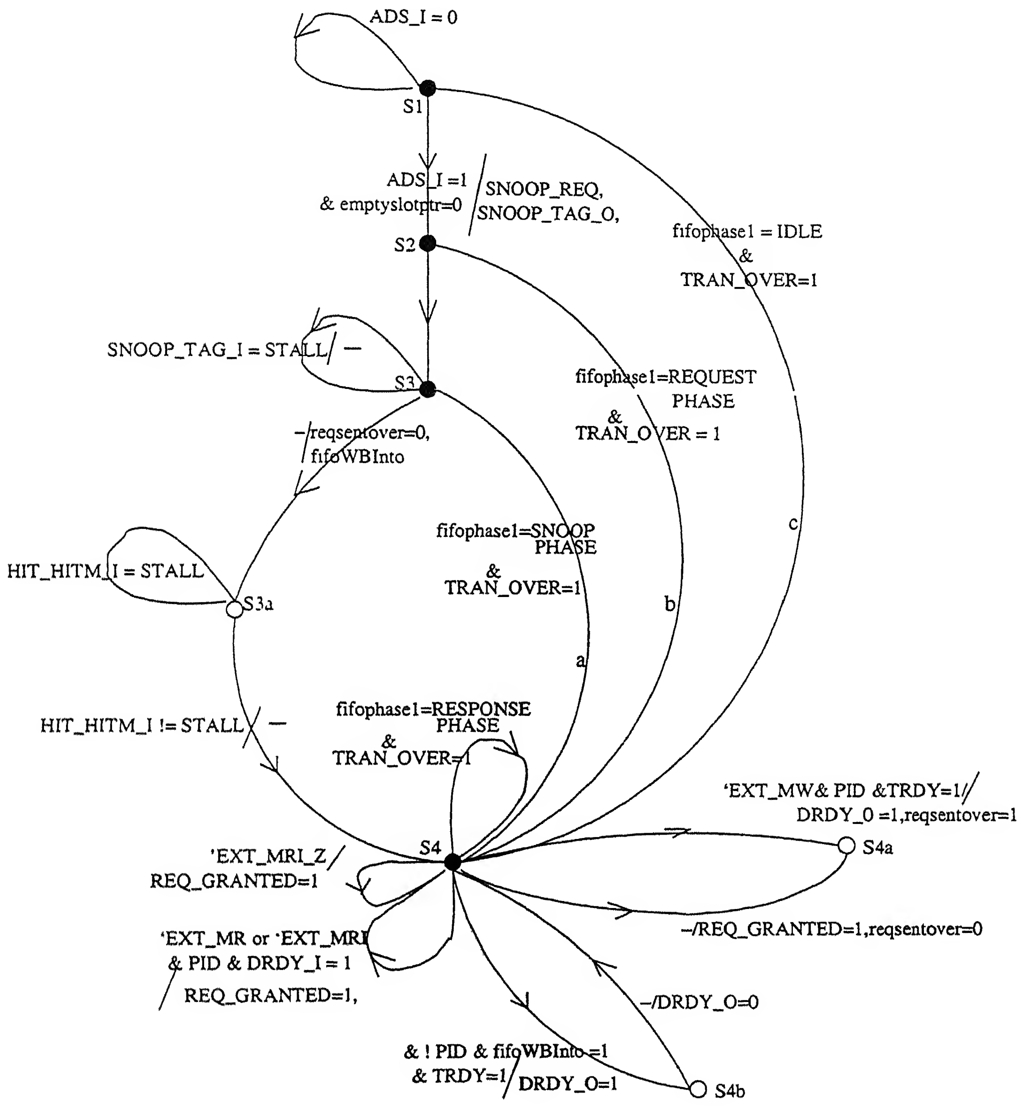


Figure 2.10: **FSM1 of Request Agent corresponding to transaction 1.**



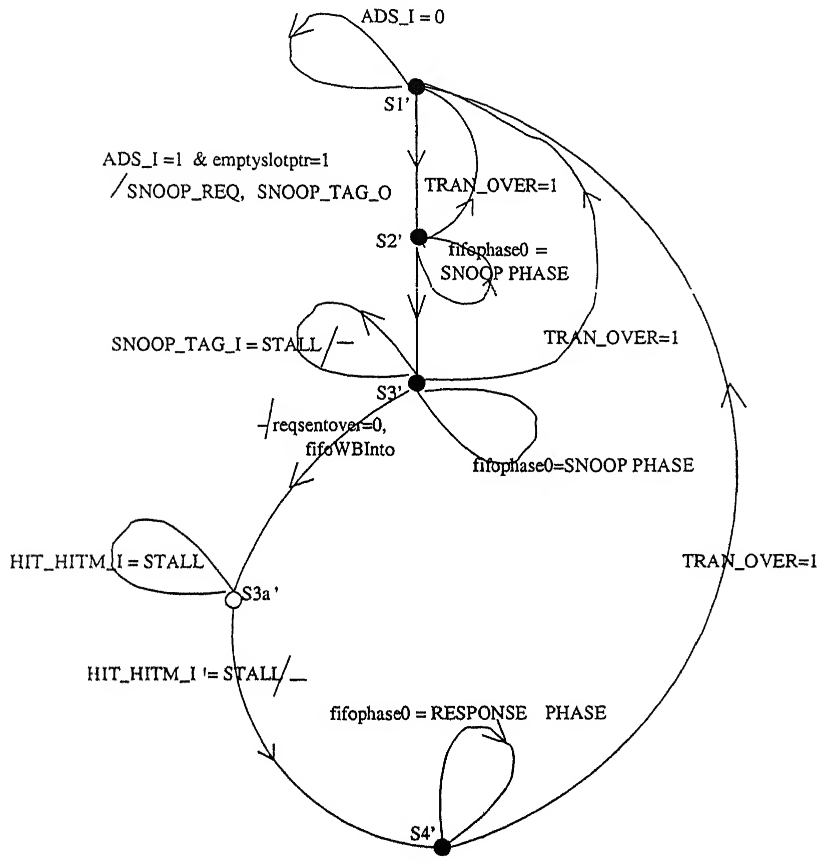
ADS_I = 0
S1
ADS_I =1 & emptyslotptr=0 / SNOOP_REQ, SNOOP_TAG_O,
fifophase1 = IDLE & TRAN_OVER=1
S2
SNOOP_TAG_I = STALL / –
S3
fifophase1=REQUEST PHASE & TRAN_OVER = 1
–/reqsentover=0, fifoWBInto
fifophase1=SNOOP PHASE & TRAN_OVER=1
c
HIT_HITM_I = STALL
S3a
b
a
HIT_HITM_I != STALL / –
fifophase1=RESPONSE PHASE & TRAN_OVER=1
'EXT_MW& PID &TRDY=1/ DRDY_0 =1,reqsentover=1
S4
S4a
'EXT_MRI_Z / REQ_GRANTED=1
–/REQ_GRANTED=1,reqsentover=0
'EXT_MR or 'EXT_MRI & PID & DRDY_I = 1 / REQ_GRANTED=1,
–/DRDY_O=0
& ! PID & fifoWBInto=1 & TRDY=1 / DRDY_O=1
S4b

● Main states

O Intermediate states

S1 – fifoPhase0 = IDLE phase
S2 – fifoPhase0 = REQUEST phase
S3 – fifoPhase0 = SNOOP phase
S4 – fifoPhase0 = RESPONSE phase

Figure 2.11: **FSM 2 of Request Agent corresponding to transaction 2**



● – Main states O – Intermediate states

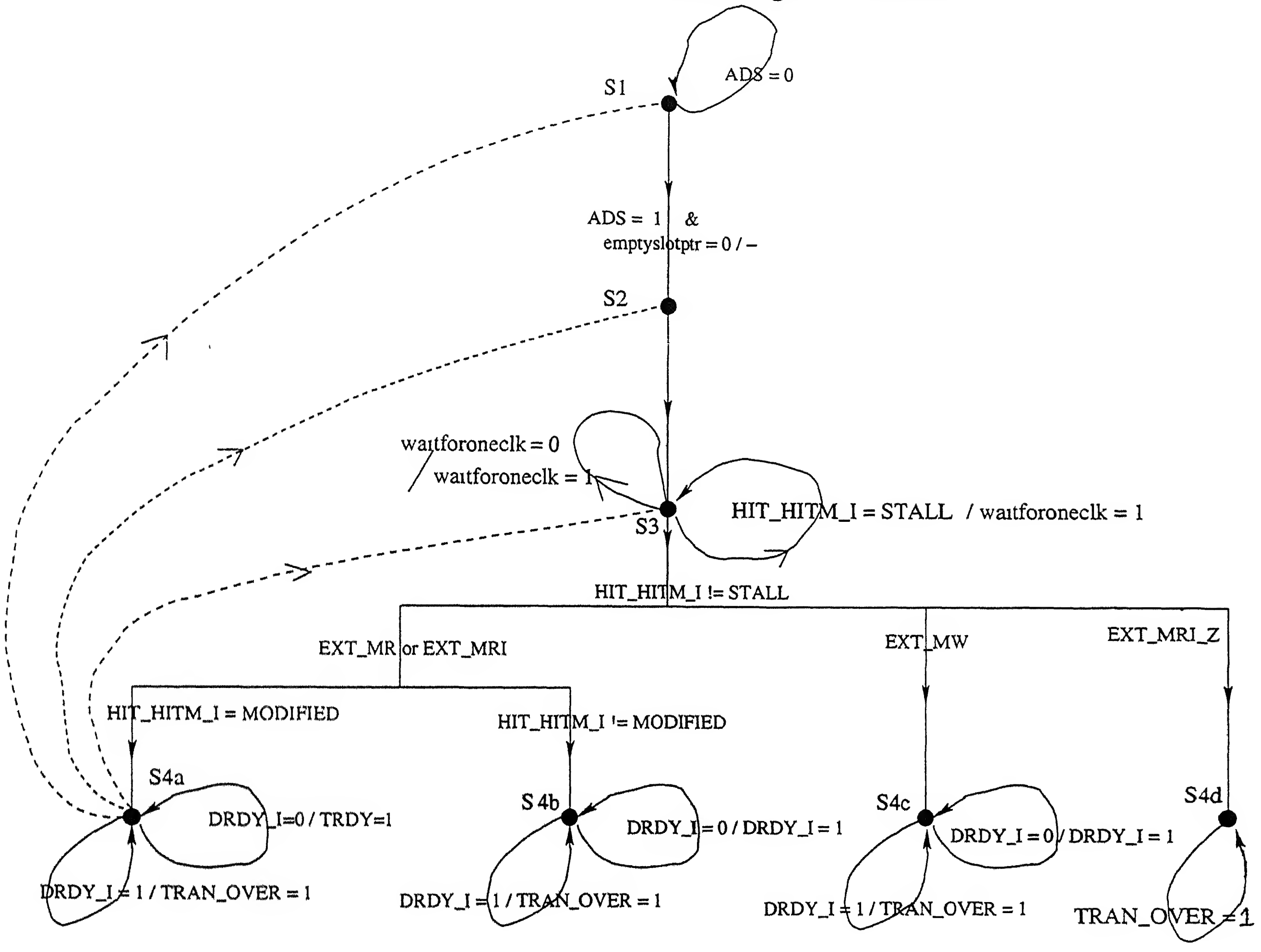
S1 – fifoPhase1 = IDLE phase

S2 – fifoPhase 1 = REQUEST phase

S3 – fifoPhase1 = SNOOP phase

S4 – fifoPhase1 = RESPONSE phase

Figure 2.12: **FSM of Memory module corresponding to transaction 1**



S1 – fifoPhase0 = IDLE
S2 – fifoPhase0 = REQUEST PHASE
S3 – fifoPhase0 = SNOOP PHASE

S4a – RESPONSE PHASE, fifoReadfromsnoop1 = 1
S4b – RESPONSE PHASE, fifowritetoreq0 = 1
S4c – RESPONSE PHASE, fifoReadfromreq0 = 1
S4d – RESPONSE PHASE

-->-- lines where TRAN_OVER is asserted and it goes to state where fifoPhase1 is in.

Note: The same transistion occurs when FSM is in S4b, S4c, S4d. But it is not shown here.

Chapter 3

Motivation and Problem definition

This chapter discusses the design issues and verification results that are reported in [1]. The chapter is organised as follows. Section 1 discusses the design issues due to the limitations of *vl2mv*. Section 2 gives the necessary background on CTL temporal logic and then presents the verification approach and results. The verification results form the main motivation of our present work. Section 3 gives the problem definition.

3.1 Design Issues

As discussed in section 1.1.3.1 of chapter 1, VIS operates on an intermediate format called BLIF-MV. VIS includes a stand-alone compiler from Verilog to BLIF-MV, called *vl2mv*. *vl2mv* supports a synthesizable subset of Verilog, and also extends it minimally to make it usable for formal verification. Given below are two features that have been added to Verilog and supported by *vl2mv*.

1. **Nondeterminism:** A nondeterministic construct, $ND, has been added to Verilog to specify nondeterminism on wire variables and is the only legal way to introduce nondeterminism in VIS.

2. **Symbolic variables:** Sometimes it is desirable to specify and examine the value of some variables symbolically, rather than having to explicitly encode them. *vl2mv*

extends Verilog to allow users to declare symbolic variables using an enumerated type mechanism similar to the one available in the C programming language. As an example, a symbolic type named *switch* is given below:

typedef enum {ON, OFF} switch;

As *vl2mv* accepts a subset of Verilog , the design code should be modified so as to be acceptable by *vl2mv*. Some of the modeling issues due to the limitation of *vl2mv* and the indirect approach taken in [1] are listed below.

- **Modeling *inout* signals**
  Some of the signals in the design are *inout* pins. As *vl2mv* doesn't support inout, the signals are explicitly separated as input and output.

- **Modeling wire-or signals**
  There are some signals which are *wire-or* ed like HIT_HITM_I. However, such signals are not accepted by vl2mv. They need to be explicitly *wire-or* ed as follows:
  assign $HIT\_HITM\_I = (HIT\_HITM\_O\_0 \mid HIT\_HITM\_O\_1)$;
  where $HIT\_HITM\_O\_0, HIT\_HITM\_O\_1$ are the output signals of processors (respectively) and HIT_HITM_I is the wire-or input signal to them.

- **Initialising all *register* variables**
  vl2mv will reject a Verilog description containing an unspecified initial state. All the *register* variables used in the Verilog code should be initialised. If the user wants a nondeterministic initial state, it should be specified explicitly using a $ND construct.

3.2 Verification approach

3.2.1 Introduction to Temporal Logic and CTL

In this section, we briefly discuss about temporal logic and CTL. A detailed description of temporal logic and CTL can be found in [5,6,7,9].

Temporal logic expresses the ordering of events in time by means of operators that specify properties such as "p will eventually hold". There are various versions of temporal logic. One is computational tree logic (CTL). Computation trees are derived from state transition graphs. The graph structure is unwound into an infinite tree rooted at the initial state. Fig. 3.1 shows an example of unwinding a state transition graph into a tree. Paths in this tree represent all possible computations of the system being modeled. Formulae in CTL refer to the computation tree derived from the model.

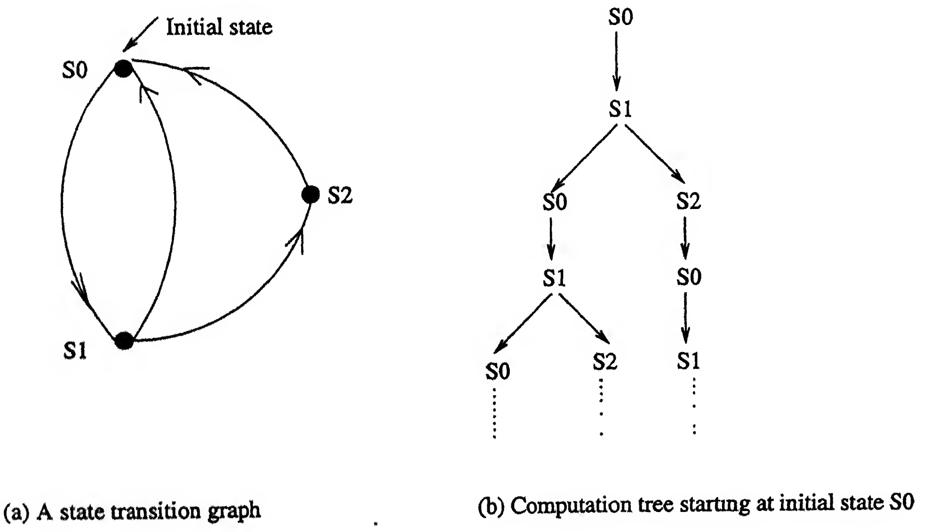

Initial state
S0
S2
S1
S0
S1
S0
S2
S1
S0
S0
S2
S1
(a) A state transition graph
(b) Computation tree startıng at initial state S0

Figure 3.1: **Computation tree from the state transistion graph**

Formulae in CTL are built from atomic propositions (where each proposition corresponds to a variable in the model), standard boolean connectives of propositional logic (e.g., AND, OR, XOR, NOT), and temporal operators. Each temporal operator consists of two parts : a path quantifier (A or E) followed by a temporal modality (F , G, X, U).

All temporal operators are interpreted relative to an implicit "current state". There are in general many execution paths (sequences of state transitions) of the system starting at the current state. The path quantifier indicates whether the modality defines a property that should be true of all those possible paths (denoted by universal path quantifier A) or whether the property needs only hold on some path (denoted by existential path quantifier E). The temporal modalities describe the ordering of events in time along an execution path and have the following intuitive meaning:

1. **F** ϕ (reads " ϕ holds sometime in the future") is true of a path if there exists a state in the path where formula ϕ is true.
2. **G** ϕ (reads " ϕ holds in the next state") is true of a path is true of a path if ϕ is true at every state in the path.
3. **X** ϕ is true of a path if is true in the state reached immediately after the current state in the path.
4. ϕ **U** ψ is true of a path if ψ is true in some state in the path, and ϕ holds in all preceding states.

Temporal logic formulas can be difficult to interpret, so that a designer may fail to understand what property has been actually verified. Therefore it is important to be familiar with the most common constructs of CTL used in hardware verification [8,9]. As an example, we discuss below a temporal logic formula which we have used in our work.

- ***AG(req $\longrightarrow$ AF ack)***
  For all reachable states (AG), if *req* is asserted in the state, then always at some later point (AF) we must reach a state where *ack* is asserted. AG is interpreted relative to the initial states of the system. AF is interpreted relative to the state where *req* is asserted. In other words, it is always the case that if the signal *req* is high, then eventually *ack* will also be high.

Fairness constraints in CTL:

In our work, we have used the fairness constraints [7,20], supported by CTL. CTL supports only *Buchii type* of fairness constraints. A *Buchii type* of fairness condition on a FSM is characterised by a subset of the state space wherein the states are visited infinitely often. A fairness constraint says that only certain “good” infinite behaviours of the design should be considered during verification; any other behaviours should be ignored as if they do not exist. The need for fairness constraints arises from non-determinism in the abstract specification of design. For example, in designing a fair scheduler for a set of processes, the scheduler may initially be specified as non-deterministically choosing which process to activate at each instant. However, we may not want to permit the behaviour where a given process is never activated. Thus, fairness constraints allow us to limit verification to those behaviours of the design where each process is activated infinitely often.

3.2.2 CTL properties

In [1], VIS simulator is initially used to catch the first level bugs in the design. Then, the model checker is used to find some intricate bugs in the design. The following important properties are identified for verifying the cache coherence protocol for the split transaction P6 bus:

1. **Liveliness**
   Intuitively, this property says that, the processor should not stall indefinitely. The cycle of execution should not stop. More formally,

   AG((Processor.stage=Fetch) $\longrightarrow$ AF(Processor.stage=Execute))

   AG((Processor.stage=Execute) $\longrightarrow$ AF(Processor.stage=Fetch)).

2. **Cache Coherency**
   If the the tag address of the cache lines in different processors match, then the cache line states should be coherent as per MESI protocol.

More formally,
AG(((No request pending)*(TLB addresses same)) $\longrightarrow$

(cacheLines I) or
(ONLY 1 cacheLine E/M and REST I)))
Note : Check only those cache line whose TLB addresses match

3. **Transaction order**
   Responses sent are in the order of the requests made. In other words, the processor requesting earlier should get the result earlier.

   More formally as,
   AG((not IOQ empty)$\longrightarrow$
   (Transaction phases in FIFO-order)).

   This CTL property is simplified in [1] as

   AG(!((processor0 in REQUEST Phase) & (processor1 in REQUEST Phase)));
   i.e., only one processor can be in the *Request* phase.

3.2.3 Verification results

In [1], it has been reported that the verification of the Liveliness, Cache Coherency and Transaction order could be successfully verified only for those cases where pseudo inputs (the opcodes of two processors and the memory address) are kept fixed.

The design model for the complete system (two processor case) is given in Table 1.

Table 1: Model Size	
Design Parameters	Components
combinational	5721
latches	298
edges	15114

Depending on the number of pseudo inputs, three different cases arise:

- **Trivial Case**
  In this case, the opcodes of the processors and the memory address variable are kept fixed. *Table 2* and *Table 3* gives the results for different combination of opcodes of processor. *Table 2* shows the result when memory addresses map to different cache lines. *Table 3* shows the result when memory addresses map to same cache lines.

Table 2 Model with fixed opcodes and Memory ≐$ND(0,1)

Config.[1]			MC[2]time (sec)			FSM	Reach.	Bdd	Analysis
L	S	LS	ioq	live	ccp	depth	States	Size	time (sec)
		1,2	43.7	516.4	44.0	62	2823	87459	158
	2	1	61.1	399.7	61.2	62	2743	115001	210
1,2			32.7	552.8	32.1	49	2445	91027	84
	1,2		14.3	127.2	14.6	46	449	31194	33
1	2		-	-	-	86	3910	47329	408
1		2	-	-	-	62	2743	65888	408

[1] L=1, S=2 implies that the opcode of Processor 1 has been fixed to LOAD and that of 2 to STORE. Similarly others.

[2] Model Checking time for properties ioq (Transaction order), live (Liveliness), and ccp (cache coherency)

– the time information for these cases in not given in [1].

Table 3: Model with fixed opcodes and Memory =$ND(0,2)

Config			MC time (sec)			FSM	Reach.	Bdd	Analysis
L	S	LS	ioq	live	ccp	depth	States	Size	time (sec)
		1,2	17.7	49.9	18.1	59	1045	42908	62
	2	1	21.5	38.3	21.6	67	1124	55825	84
1,2			24.1	211.4	23.9	72	2398	73822	147
	1,2		9.1	12.1	9.4	43	317	16244	17
1	2		12.4	20.0	12.8	74	1327	15866	313
1		2	15.5	37.7	16.7	67	1124	40941	205

- **Non-Trivial Case**

  In this case, the opcode of one processor is kept fixed and the other processor opcode is kept variable. The memory address is chosen such that memory addresses map to same cache lines. *Table 4* gives the result for some cases in which verification was possible.

Table 4: Model with one variable opcode
and Memory Address =$ND(0,2)

Config			MC time (sec)			FSM	Reach.	Bdd	Analysis
L	S	LS	ioq	live	ccp	depth	States	Size	time (sec)
1	1	2	-	-	-	126	93185	61203	1534
2	1	1	-	-	-	145	118811	91761	2150

- **Impossible Case**

  In this "general" case, the opcodes of both the processors are kept variable. The memory address is chosen such that memory addresses map to same cache lines. In [1], this case could not be verified. Only a partial reachable analysis was reported whereas full reachability was reported to be impossible.

  - Reachable States (at depth 24) = 1.16×10^7

    Bdd Size = 1,016,521

The results show that as we increase the degrees of freedom, there is an explosion of states. The model checking time also increases exponentially. Even though the model is simplified, without compromising on the complexity of protocol, explosion of states has occurred (state-explosion problem) for the general case (where the opcodes of both the processors are kept variable). It was conjectured in the report that this could be minimised using compositional verification techniques.

3.3 Motivation and Problem definition

As discussed above, the inability to verify the "general" case (the opcodes of both the processors are kept variable) due to state explosion, motivated us to undertake the challenging task of formally verifying the "general" case using compositional verification techniques.

More formally, the problem definition:

To formally verify the cache coherence protocol for the "general" case by employing compositional verification techniques.

The next chapter discusses the compositional verification approaches reported in the literature.

Chapter 4

Compositional Verification techniques

The need to resort to compositional verification approaches arises from several reasons. Higher device integration has spurred the growth of system on chip (SOC) designs. With reduced product life times, the pressure to market on time and the need to avoid costly design iterations have led to the demand that the designs be successful on the first silicon. Formal verification based on symbolic model checking has been employed successfully in such prefabrication verification efforts. The problem of state and memory explosion present in the symbolic model checking approach, limits the size of designs that can be verified without any modifications to the RTL code. SOC designs being inherently large, need to be verified using a divide and conquer approach. Generally, such an approach is taken intuitively by designers, based on their detailed knowledge of a design. A formalization of verification approaches for large systems, such as System on Chip (SOC), is being attempted by several research groups with a view to automate the process. In the literature, such approaches are referred to as *compositional verification* [10 - 14]. In [15,16], Assume - Guarantee approach, a compositional verification technique was successfully used to verify Audio Output Interface (AOF) module and Picojava External Bus Interface (EBI) using a state of art symbolic model checking tool, BINGO.

The chapter discusses an overview of two compositional verification approaches reported in the literature.

4.1 Compositional verification

We refer to [10,11,12] for detailed discussions on compositional verification approaches.

The state explosion problem observed in the verification of large systems using symbolic model checking, arises due to the parallel composition of finite state modules. The severity of the problem is strongest for loosely coupled modules which communicate through fewer shared variables. This limits the size and complexity of a system that can be verified by symbolic model checking. Most design implementation of systems are carried out heirarchically through an interconnected set of well defined components or modules. Thus, there exists natural boundaries in any design implementation which can be exploited to overcome the state explosion problem. One obvious approach is to reduce the number of modules involved in the verification effort to a value which can be easily handled by the symbolic model checker with respect to the available computational resources. This divide and conquer approach, when taken, will force a decomposition of the specifications, or properties, of a system, into specifications, or properties, of its modules. These module properties can be verified seperately. This, however, enforces upon us the obligation of proving that the module specifications do imply the specifications of the entire system. This is called *Compositional Verification*. In the figure below, we have a system composed of two finite state modules P and Q, which communicate with each other over channels (or wires) S, and also with their external environment (represented by the dashed vertical lines).

Let the parallel composition of P and Q be given by, $P \parallel Q$. If the module specifications, or properties, of P and Q are φ_P and φ_Q respectively, we would like to use the following inference rule:

If

$$P \models \varphi_P,$$

$$Q \models \varphi_Q, \quad and$$

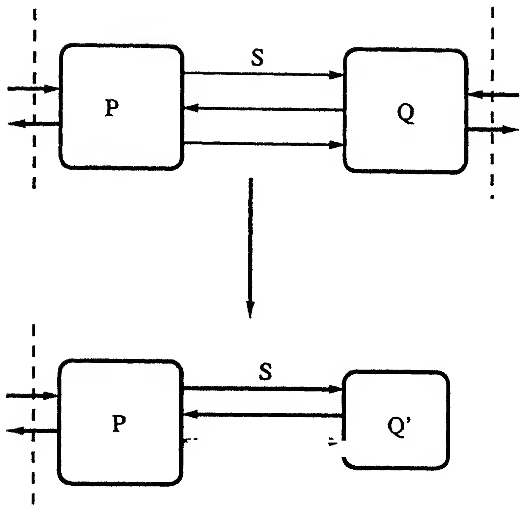


Figure 4.1: **Compostional Minimization**

$$\varphi_P, \ and \ \varphi_Q \ \Rightarrow \ \varphi$$

then

$$P \parallel Q \models \varphi.$$

In general, a component/module of any system is usually optimized and implemented to function only in the environment of that system. Therefore, in general, P will not satisfy φ_P in any arbitrary environment. Any unexpected input that it receives, will cause P to traverse an unintended path in its state space, consequently, resulting in an incorrect behaviour. The reason for this is the fact that, the reachable state space of P, in isolation, can be much larger than the state space of P composed with Q. [11,12] refer to this as the *Environment Problem.*

There are two approaches to solving the *Environment Problem.* These are

- Compositional Minimization.
- Assume-Guarantee Reasoning.

4.1.1 Compositional Minimization

In *Compositional Minimization*, an equivalent abstraction $\grave{Q}(\grave{P})$, of $Q(P)$, as shown in Fig. 4.1, is extracted, with respect to its behaviour observable by $P(Q)$, over the channels (wires) connecting them. This abstraction is called an *Interface Process*. By abstracting out the communications of $Q(P)$ invisible to $P(Q)$ can result in an *Interface Process*, $\grave{Q}(\grave{P})$, with very few states. $\grave{Q}(\grave{P})$ can be easily obtained by minimizing the finite automaton representing $Q(P)$. An *Interface Rule* based on the above approach can be stated to contain the state explosion problem in concurrent systems.

The interface rule states that if :

- $P \equiv \grave{P}$ on the set S of wires/channels (Fig. 4.1),
- φ is a CTL formula whose atomic propositions denote signals of Q, and
- φ is true in $\grave{P} \parallel Q$ (the composition of $\grave{P}$ and Q),

then φ is true in $P \parallel Q$.

In a loosely coupled system, $\grave{P}$ will almost always have fewer states than P, and therefore, $\grave{P} \parallel Q$ will be much smaller than $P \parallel Q$. This rule can be easily extended to boolean combinations of CTL formulas.

4.1.1.1 Disadvantages and Advantages of the approach

In many loosely coupled systems, it is possible that $\grave{P}$ is as complex as P itself on the set S of wires/channels. In such instances, this approach will not offer any benefits. However, for instances which are not beset by this problem, the approach can be very successfull. It also lends itself to a higher degree of automation as compared to the *Assume-Guarantee Reasoning*, and is fully applicable to temporal properties based on CTL.

4.1.2 Assume-Guarantee Reasoning

In *Assume-Guarantee Reasoning*, the *Environment Problem* is deferred by expressing any assumptions process P makes about process Q in a temporal formula ψ_Q.

These assumptions can be used when checking that P satisfies its specification φ_P. These assumptions must, of course, be discharged relative to Q. To do this, we use the inference rule given below.

If

$$P \parallel \psi_Q \models \varphi_P, \quad and$$

$$Q \models \psi_Q$$

then

$$P \parallel Q \models \varphi_P$$

By $P \parallel \psi_Q \models \varphi_P$, we imply that $P \parallel \grave{Q} \models \varphi_P$, for all $\grave{Q}$ such that $\grave{Q} \models \varphi_Q$, .

Assume-Guarantee Reasoning is, therefore, a technique that enables verification of each module individually. As the behaviour of each module depends on that of the rest of the system, i.e. its environment, the user must specify properties that the environment needs to satisfy in order to guarantee the correct behaviour of the component. These properties are *assumed*, and if satisfied, ensure correct behaviour, if any, of the module, with respect to its specifications, called the *guarantees*. By properly formulating the set of *assume* and *guarantee properties*, it is possible to demonstrate the correctness of the entire system without constructing the global state graph.

4.1.2.1 Disadvantages and Advantages of the approach

Though, we demonstrate(in the next chapter) the feasibility of using the *Assume-Guarantee Reasoning* in the verification of liveliness property to the cache coherence protocol for the split transaction P6 bus , it may not be the case for every design. The success of this approach has been reported only with respect to those properties which belong to the class of temporal properties known as ACTL[1], a subset of CTL. For a given set of interacting modules, it may not always be possible to decompose the given global

[1] ACTL is the subset of CTL containing just those formulae which have no E path quantifiers when put in positive normal form.

property into sub-properties belonging to ACTL. As will be clear, this approach needs detailed design information for its success and is therefore, not easy to automate. This limits it usage by a wider community, as only the design team has access to such information. However, for instances where it is applicable, it offers the largest possible benefits in terms of reduced computational complexity with respect to space and time.

Chapter 5

Verification approach using Assume-Guarantee technique

In [1], it was shown that for the abstracted model of the P6 multiprocessor configuration, model checking for the case where the opcodes of both the processors are kept variable and variable memory addresses mapping to same cache lines cannot be verified by symbolic model checking. This is because as the degrees of freedom is increased state explosion problem is encountered. Besides the size of the BDD also becomes too large. The model checking time also increases exponentially. This case was classified as an 'impossible case'. It was conjectured that this 'state explosion problem' can be minimized by employing Compositional Verification techniques.

In this chapter, an approach based on compositional venfication technique known as *Assume and Guarantee* appraoch is presented for the above P6 multiprocessor configuration with respect to two properties, the liveliness and Transaction order property (see section 3.2.2). It is shown that these properties can be verified using compositional verification techniques.

5.1 Verification Approach

Each processor has three modules, Execution Unit, Cache and Request agent. Each module is a sequential circuit which can be represented as an individual FSM. These FSM's interact with each other as shown in Fig. 2.6. For example, Request Agent interacts with Cache, Processor1 and Memory. It is through the Request Agent that a Processor communicates with another processor and memory. This is as shown in Fig. 2.7. In each Processor there are three interacting FSM's . Thus, when we consider the system as a whole, these FSM's belonging to Processor0, Processor1 and Memory interact with each other. The individual FSM's in each module of a processor are shown in Fig. 2.3, 2.8, 2.10, 2.11 and 2.12 .

The liveliness property is an important property to check. The property states that the Processor should not stall indefinitely. More formally,

AG((processor.stage = FETCH) $\longrightarrow$ AF(processor.stage = EXECUTE))
and
AG((processor.stage = EXECUTE) $\longrightarrow$ AF(processor.stage = FETCH))
should be satisfied.

We will define two terms, *main machine* [13] and *side machine*[13] which we have used while describing the verification approach. *Main machine* refers to the FSM on which we are verifying the temporal logic prpery. FSM's which interact with the main machine are called *side machines*. In Fig. 4.1, P is the *main machine* and Q is the *side machine*. The side machines constitutes the *external environment* of the main machine.

Verification of each of the above two CTL properties involves the signals of a single large machine say, Processor0, interacting with other machines (Memory and Processor1) which constitutes its environment. Hence, the othe (side) machines, Processor1 and Memory can be minimised by hiding the output variables which the main machine (Processor0) cannot observe e.g., the internal registers of Processor1 and Memory. The behaviour related to the output variables of the side machines with which they interact with the main machine are either modeled by smaller FSM's, which are the extracted

FSM's of the side machines or they are driven by smaller FSM's based on some assumptions which in turn become the guarantees for the side machines.

5.1.1 First Approach: Processor0 as main machine

In the first approach, we partitioned the entire system as three large large machines (FSM's), i.e., Processor0, Processor1 and Memory. Since Processor0 and Processor1 are identical designs, the liveliness property can be checked on either Processor0 or Processor1. The Processor0 was chosen for verification.

Processor0 interacts with Processor1 and Memory through the Request Agent. This is shown in Fig. 2.7. The external inputs to Processor0 from the external environment (Processor1 and Memory) have to be constrained. Some of the inputs to Processor0 from the external environment do not affect the verification of a particular property. These inputs can be fixed to constant values. These fixed inputs will not contribute to the product FSM of the entire system thereby reducing the number of states. For example, transactions over the data bus does not affect the verification of the Liveliness Property. But, it can affect the Cache Coherance Property. In the verification of the Liveliness Property on Processor0, the outputs EXT_DATA_O_0, EXT_DATA_O_1 and EXT_DATA_O_M are all fixed to 0. This results in EXT_DATA_I which is an input to Processor 0 to become 0. However, extreme care must be exercised while constraining the inputs to constant values. This is because, it is possible to validate a wrong property by assigning values to these inputs which do not hold in the actual environment.

5.1.1.1 Assumptions on external inputs

***ADS_O_1* :**

ADS_O_0 from Processor0 and ADS_O_1 from Processor1 are wired-ORed to generate ADS_I which goes as a input to Processor0 (Fig. 2.7). The external output signal ADS_O_1 from Processor 1 is constrainled in the following way. When ADS_O_1 is asserted, Processor1 has already acquired the address bus and its transaction is placed in the FIFO buffer. The request from Processor 1 may come at any point of time once certain conditions are satisfied. However, the request signal is sampled only at the rising edge of the clock. The conditions under which ADS_O_1 is asserted are as follows.

1. If BREQ0 = 1, then ADS_O_1 cannot be asserted according to the '*built-in rotation priority*' scheme given in chapter 3.
2. If ADS_O_0=1, then ADS_O_1 cannot be asserted as the Processor 0 is also acquiring the bus at the same time.

If these two conditions are satisfied, then ADS_O_1 can be made high. Hence, ADS_O_1 is non-determinised once these conditions are satisfied. It is very important to note that non-determinising the input variables will cause the state transistion graph to be traversed for all possible values of the input. If a CTL property passes for a non-deterministic case, it is guaranteed to pass for the real external environment in which the main machine is placed. A simple FSM for the generation of ADS_O_1 is given below in Fig. 5.1.

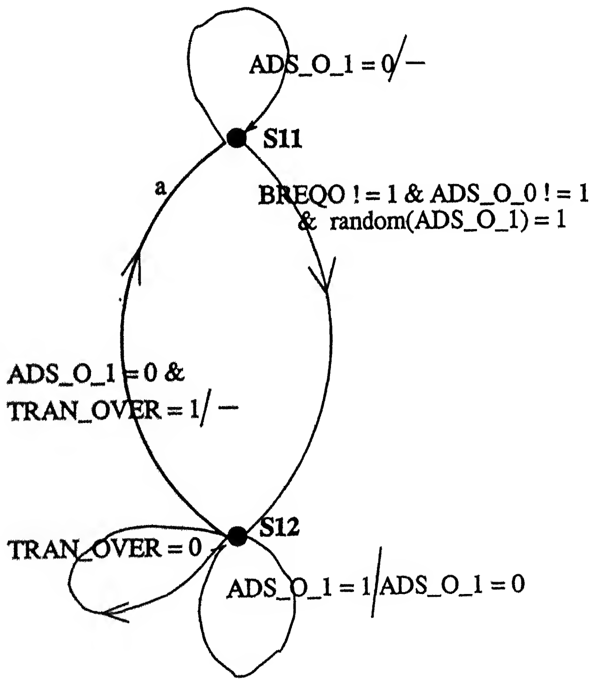


Figure 5.1: **FSM that constrains ADS_O_1**

The above abstracted FSM is extracted from the large FSM of Processor 1. When ADS_O_1 is asserted, it goes from initial **S11** state to *S12*. In the next clock cycle, ADS_O_1 is deasserted and remains in S12. It goes back to **S11** after the transaction is completed (TRAN_OVER=1). In the above FSM, ADS_O_1 is non-determinised again once it reaches **S11** from **S12**. But, this results in the same situation as it was before ADS_O_1 is asserted. So, the above FSM can also be constrained by removing the edge 'a' going from **S12** to **S11**. As a result of this, FSM depth and number of reachable states are reduced. It is very important to note that this way of constraining the input is property dependant.

***EXT_TRANS_REQ_O_1 and EXT_ADDR_0_1* :**

The Processor1 outputs EXT_TRANS_REQ_O_1, TID_O_1 and EXT_ADDR_O_1 can be easily constrained. When the Processor 1 sees ADS_O_1 asserted, it puts the following information on the bus, Request type of the transaction (EXT_TRANS_REQ_O_1), address of the cacheline requested(EXT_ADDR_O_1) and ID of the Processor(TID_O_1) which acquired the bus. The processor 0 which is snooping over the bus, latches these signals when ADS_I goes high (as ADS_O_1 goes high). Hence, when *ADS_I = 1*, EXT_TRANS_REQ_O_1 can take any of the request types, EXT_MR, EXT_MRI, EXT_MW and EXT_MRI_Z. EXT_ADDR_O_1 can take values either 0 or 2. In [1], the property was proven only for the case where the pseudo input corresponding to opcode of one processor is kept fixed, and for memory address mapping onto same cache lines. We prove the property for a more general setting by removing the restriction on fixed opcodes. In our case the opcode can be variable for both the processors.

TID_O_1 is fixed to 1 (PID1), as the Processor1 ID is 1. EXT_TRANS_REQ_O_1 and EXT_ADDR_O_1 are reset to the initial values in the next clock cycle.(as they are latched inside into the FIFO buffer). The initial values set for EXT_TRANS_REQ_O_1 and EXT_ADDR_O_1 are NOREQ and 0 respectively. A simple FSM which implements this is given in Fig. 5.2.

***HIT_HITM_O_1* :**

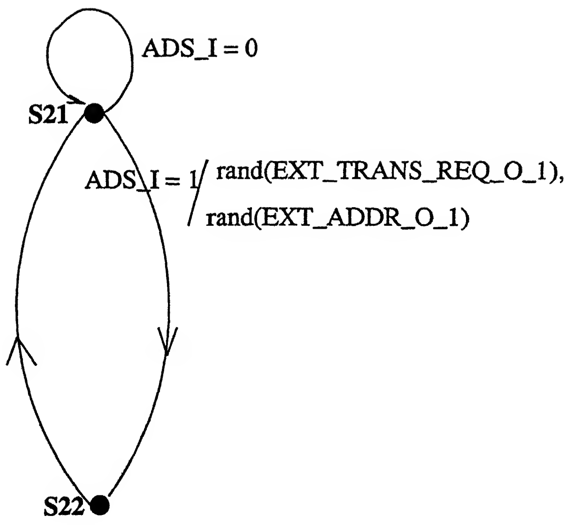


Figure 5.2: **FSM that constrains inputs EXT_TRANS_REQ_O_1**

HIT_HITM_O_0 from Processor 0 and HIT_HITM_O_1 from Processor1 are wired-ORed to generate HIT_HITM_I which goes as a input to Processor 0 (Fig. 2.7). In our approach, we need to constrain the external signal HIT_HITM_O_1 generated by Processor1. After processor0 enqueues its transaction in the FIFO queue, it proceeds from REQUEST_PHASE to SNOOP_PHASE (**S2** to **S3** or **S2'** to **S3'**) as shown in Fig. 2.10, 2.11. Then, it proceeds either to **S3a** or **S3a'**. It stalls in the new state if HIT_HITM_O_1 (HIT_HITM_I) is equal to STALL indefinitely as shown in the state transistion graph in Fig. 2.10 and Fig. 2.11. In this state, HIT_HITM_O_1 can take any random value. But, this also includes the case where it could be equal to STALL indefinitely. The real environment with which it interacts, however, may not generate this case. Hence, while verifying the liveliness property for the Processor0, we have to impose a fairness constraint on HIT_HITM_O_1 so that it will eventually come out of the stall condition. Thus, the external environment (Processor1, in this case) must satisfy the fairness constraint as a guarantee.

In CTL language, the fairness constraint on HIT_HITM_O_1 can be written as

! (HIT_HITM_O_1 = STALL);

A simple FSM which implements this is given below.

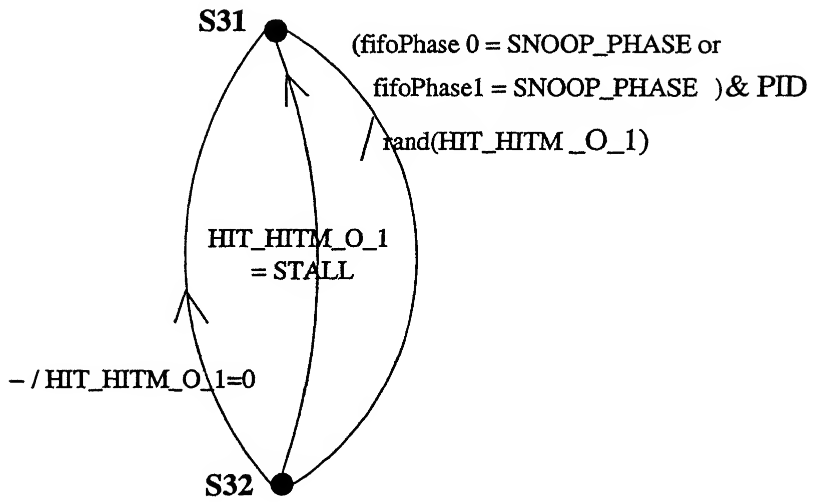


Figure 5.3: **FSM that constrains HIT_HITM_O_1**

TRDY **:**

TRDY is the output signal generated from MEMORY which goes as a input to Processor0 as shown in Fig. 2.7. TRDY is asserted under the following conditions.

1. When the transaction is in RESPONSE_PHASE and request type is EXT_MR or EXT_MRI and when there is a hit on modified line, *HIT_HITM_I = MODIFIED*.

When the transaction is in RESPONSE_PHASE and request type is EXT_MW.

DRDY_O_1 **:**

DRDY_O_0 from Processor 0, DRDY_O_1 from Processor 1 and DRDY_O_M from MEMORY is wired_ORed to generate DRDY_I. If the cache line requested by Processor0 is present in Processor1, then DRDY_O_1 is asserted otherwise DRDY_O_M is asserted by Memory. Therefore, we can logically OR the two signals DRDY_O_1 and

DRDY_O_M to generate DRDY_O_C. DRDY_I can be seen to be a wired-OR signal of DRDY_O_0 and DRDY_O_C. Now, we need to constrain DRDY_O_C. If the Processor0 has as the state of the transaction the REQUEST_PHASE, and the Request type is EXT_MR or EXT_MRI then we can assume that DRDY_O_C will be asserted (either by Memory or Processor1). After the Processor1 transaction has entered the FIFO queue and if the cacheline requested by the Processor1 is not present in Processor0, then Memory asserts DRDY_O_M. This in turn will result in DRDY_O_C being asserted. Thus, DRDY_O_C is asserted for the following conditions.

1. If Processor0 is in RESPONSE_PHASE and Request Type is EXT_MR or EXT_MRI.

2. If Processor1 is in RESPONSE_PHASE and Request Type is EXT_MR or EXT_MRI and the cacheline is not present in Processor0 (*fifoWBInto=0*). The above conditions can be treated as assumptions. These assumptions can be easily extracted from the STG of the memory module shown in Fig. 2.10.

TRAN_OVER :

TRAN_OVER is an output signal generated by Memory. TRAN_OVER is asserted under the following conditions (Fig. 2.12.).

1. If processor 0 is in RESPONSE PHASE and Request Type is EXT_MRI_Z.

2. If DRDY_I is asserted, then TRAN_OVER is asserted in the next state (clock cycle).

These conditions can be extracted easily from the STG of Memory as shown in Fig. 2.12.

With the external environment modeled as above, we Liveliness property due to the *state explosion problem* . This is because the number of states present in the product FSM obtained from the main machine (Procesor0) and the individual FSM's driving each input of Processor0 is very large.

5.2 Second approach: Request agent of Processor0 as main machine

In our next approach, we partitioned the Processor0 module into two sub-modules, the Request Agent and a combination of the Execution unit and the cache module. The decoupling of the module interconnections necessitates recasting the original liveliness property in terms of the properties of the decoupled modules. This requires a detailed analysis of design and a detailed knowledge of the state transistion graph. Reconsider the liveliness property on Processor0.

Property1:
AG((processor.stage = FETCH) $\longrightarrow$ ***AF(processor.stage = EXECUTE))*** ;
Property2:
AG((processor.stage = EXECUTE) $\longrightarrow$ ***AF(processor.stage = FETCH))*** ;

The information in the state transistion graph can be very useful in splitting the original property on the decomposed modules. Consider the state transistion graph of Execution Unit in Fig. 2.3. From the initial state, executionstage = FETCH, it proceeds to the EXECUTE stage in the next cycle without any conditions on the inputs. Therefore, Property1 given above will surely pass. For the second property to pass, it requires CACHE_RDY or DATA_RDY to be asserted by the cache unit. Now, consider the STG of cache unit in Fig. 2.8. If transreqpending = 1 (INT_TRANS_REQ != 0), then DATA_RDY will be asserted only if REQ_GRANTED = 1. This can be seen in the fig 2.8 with respect to the edges drawn with thick lines. Therefore, the original liveliness property :

AG((processor0.stage = EXECUTE) $\longrightarrow$ ***AF(processor0.stage = FETCH))***

can be *restated* with respect to the Request Agent module of Processor0 as follows.

Property3:
AG((INT_TRANS_REQ ! = NOREQ) $\longrightarrow$ ***AF(REQ_GRANTED = 1))***

5.2.1 Assumptions and Guarantees on Request agent

5.2.1.1 Assumptions on external signals

We now have to set up the environment for the Request Agent of Processor0. It can be seen that the Request Agent of Processor0 interacts with Cache block, Memory and Processor1. The signals through which the Request Agent interacts with Processor1 have already been constrained as discussed in our first approach. We only need to constrain the signals through which the Request Agent interacts with the Cache block.

INT_TRANS_REQ and INT_ADDR :

INT_TRANS_REQ and INT_ADDR are the outputs signals generated by the Cache block. These are input signals to Request agent. INT_TRANS_REQ is non-determinised and can take any of the following values, NOREQ, EXT_MR, EXT_MW, EXT_MRI and EXT_MRI_Z. If INT_TRANS_REQ is a valid request type (not equal to NOREQ), then INT_ADDR can take either 0 or 2. They values are latched inside the Request agent and then in the next clock cycle both are reset to initial values of NOREQ and 0 respectively. A simple two state FSM can be designed to implement this behaviour as shown in Fig. 5.4. In the STG shown below, after going to state **S42**, INT_TRANS_REQ is deasserted to NOREQ and then waits for REQ_GRANTED to go high, when this happens it goes to **S41** and the process repeats. In the above FSM, INT_TRANS_REQ is non-determinised again once it reaches **S41** from **S42**. But, this results in the same situation as it was before INT_TRANS_REQ is asserted. So, the above FSM can also be constrained by removing the edge '**b**' going from **S42** to **S41**. As a result of this, FSM depth and number of reachable states are reduced. It is very important to note that this way of constraining the input is property dependant.

SNOOP_TAG_I :

As shown in Fig. 2.6, Request Agent interacts with the Cache unit through the input SNOOP_TAG_I. When the Processor1 transaction is enqueued in the FIFO buffer of Processor0, SNOOP_REQ is asserted. With SNOOP_REQ asserted, the Cache Unit sends

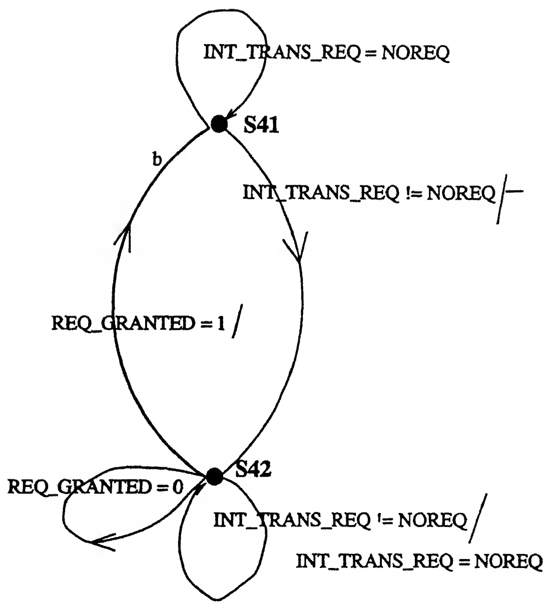


Figure 5.4: **FSM that constrains INT_TRANS_REQ**

the state of the cacheline in Processor0 through the SNOOP_TAG_I. If the Cache is busy updating the same cache line, then SNOOP_TAG_I = STALL is returned. When SNOOP_REQ is asserted, SNOOP_TAG_I is allowed to take any random value. If *SNOOP_TAG_I = STALL*, then SNOOP_REQ is asserted. Hence, the transaction stalls if SNOOP_TAG_I returns STALL indefinitely. For this case, as shown in the Fig. 2.10, 2.11, STG remains in S3 or S3' and the CTL property will fail. This situation may not arise in the real environment. So, the property should be verified on only fair paths. In CTL, the fairness constraint imposed on the input SNOOP_TAG_I can be written as

! (cache.SNOOP_TAG_I = STALL)

This fairness constraint therefore becomes a guarantee for the Cache block. In other words, the Cache block should guarantee that this stall case will never occur. Although the Property3 mat pass for the Request Agent under the fairness constraint, if the fairness constraint cannot be guaranteed by the Cache block, the given property will be deemed unsatisfiable for the whole system under consideration.

INT_DATA_I and SNOOP_DATA :

The other inputs INT_DATA_I and SNOOP_DATA to Request Agent as seen in Fig. 2.7 are fixed to constant values as data transactions will not affect the verification of liveliness property. The behaviour of the Request agent which is not needed for the verification of the property are removed. For example, the behaviour of the Request Agent by which the tag of the cache lines are affected is removed. But, the same cannot be done while verifying a property related to cache coherence protocol. The output registers, INT_DATA_O, SNOOP_TAG_O and INT_TAG inside the Request agent are all fixed to constant values.

5.2.1.2 Guarantee on REQ_GRANTED

With the external environment constrained as above and with fairness constraints applied on HIT_HITM_O_1 and SNOOP_TAG_I, the CTL property

AG((INT_TRANS_REQ != NOREQ) ⟶ AF(REQ_GRANTED = 1)) ;

failed on the Request agent block. This is because of a *bug* in the code which prevents REQ_GRANTED to be asserted even if INT_TRANS_REQ = EXT_MW. This can be explained with respect to the state transistion graph shown in Fig. 2.10. Consider the situation where Processor0 enqueues its transaction in the FIFO queue and INT_TRANS_REQ = EXT_MW. Later, Processor1 also enqueues its transaction in the FIFO queue. Processor 0 is in state **S4a** (Fig. 2.10). In this state, internal signal *reqsentover* is asserted. If Processor1 transaction is in state **S3'** at the same time, then *reqsentover* is deasserted (Fig. 2.11). In other words, two FSM's driving the same register element concurrently. This results in non-determinising the signal reqsentover. If *reqsentover is 0*, the transaction belonging to Processor0 stalls in **S4** and REQ_GRANTED will never be asserted. This bug had been overlooked in [1] because the opcodes of one of the processors is kept fixed. If Proceşsor0 can only drive the register *reqsentover* in its request agent, then this error will not arise. After fixing this *bug*, the CTL property cessor1 and vice versa and also Processor0 and Processor1 are identical designs. Therefore, the variables of Request agent of Processor1 can be mapped on to the variables of Processor0. Therefore, the property can be restated on the Request Agent of Processor0 as:

Property4:
AG(!((processor0.fifophase0 in REQUEST Phase) & (processor0.fifophase1 in REQUEST Phase)));

It is very important to note that this is a very special case and it may not be true for all cases.
The above property verified on the Request agent.

5.2.2 Assumptions and Guarantees on combination of Execution unit and Cache module

The original liveliness property can be verified on the combination of Execution unit and Cache module. While verifying the property, we have to make an assumption on the input signal REQ_GRANTED. The assumption being that, if *INT_TRANS_REQ != NOREQ*, then eventually at some point of time REQ_GRANTED is asserted by the Request Agent. This assumption is already 'guaranteed' by the Request agent as discussed in the above section.

5.2.2.1 Assumptions on combination of Execution unit and Cache module

The liveliness property is to be verified on the combination of Execution Unit and Cache module. The main machine on which we are verifying the property is composed of Execution unit and the Cache module. The external environment for this constituted module is the Request agent.

SNOOP_REQ :

We have to constrain the external signal SNOOP_REQ which goes as an input to the main machine (Execution Unit + Cache). It is through the SNOOP_REQ that the Cache unit of Processor0 indirectly interacts with Processor1. If INT_TRANS_REQ is equal to NOREQ, it means the Processor0 is not requesting access to the bus and is busy, either reading a cache line or updating a MODIFIED, or EXCLUSIVE, cache line. We have randomized the SNOOP_REQ for this case. If *INT_TRANS_REQ != NOREQ*, then SNOOP_REQ may be 1 or 0 depending on whether Processor1 transaction is enqueued in the FIFO buffer. If the Processor1 had requested for the bus and its transaction is enqueued in the FIFO buffer, then SNOOP_REQ is asserted and then it will be deasserted basing on the signal output SNOOP_TAG_O coming from Cache unit of Processor0. If the Processor0 Cache unit is busy updating the Cache Memory, and the snoop request for the same cache line is requested then SNOOP_TAG_O generates STALL and SNOOP_REQ is asserted until SNOOP_TAG_O returns a valid tag. When this happens, SNOOP_REQ is deasserted. Once REQ_GRANTED=1, then process is repeated. This is clear from the two state FSM which is given below.

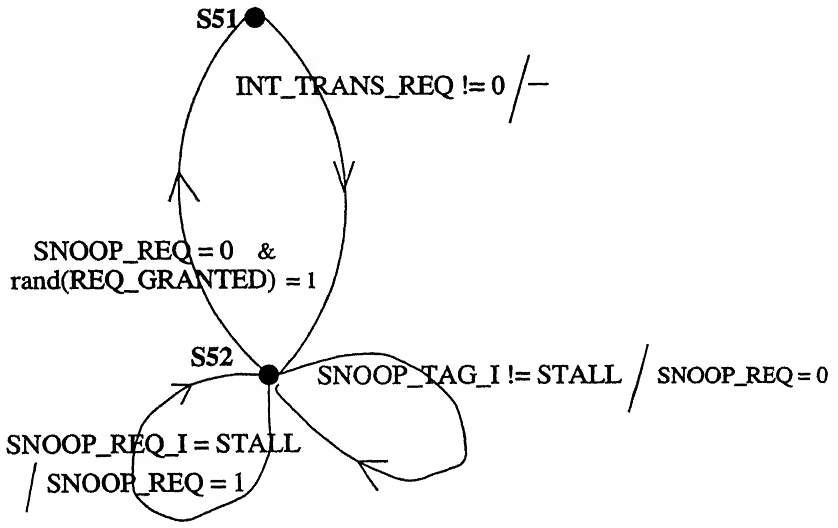


Figure 5.5: **FSM that constrains SNOOP_REQ**

Care must be exercised while constraining inputs coming from the external environment. It may not always be possible to non-determinise the variables. A small FSM behaviour have to be written to constrain these inputs. While constraining such inputs, if we impose more rigorous constraints than that imposed by the environment, then the property can fail. In that case we have to debug and find whether the failure of the property is due to wrong behaviour of the system under verification or due to wrong constraints imposed by the external environment.

***SNOOP_ADDR, INT_DATA_I, SNOOP_TAG_I and INT_TAG* :**

When *INT_TRANS_REQ != NOREQ*, then SNOOP_ADDR is allowed to take any of the values 0 or 1. The other inputs INT_DATA_I, SNOOP_TAG_I and INT_TAG are fixed to constant values. The output and internal data registers of the cache block, SNOOP_DATA, INT_DATA_O etc. are also fixed to constant values.

5.2.2.2 Guarantees

With the above assumptions and external environment constrained as above, the original liveliness properties (1,2) passed under these costraints.

***Guarantee on SNOOP_TAG_I and HIT_HITM_O_0* :**

While verifying the Request agent block, we imposed fairness constraint on SNOOP_TAG_I as discussed in section 5.2.1.1. The fairness constraint can be 'guaranteed' by the cache block by verifying the following CTL property :

Property5:
AG((processor0.CACHE.SNOOP_TAG_O = STALL) ⟶
AF(processor0.CACHE.SNOOP_TAG_O = VALID_REQ));

This Property was verified correctly for the combination of Execution and Cache module. HIT_HITM_O_0 is generated by the Request Agent depending on the value of the input SNOOP_TAG_O from the Cache unit. Since the above property guarantees that SNOOP_TAG_O does not remain in STALL indefinitely, it is guaranteed that HIT_HITM_O_0 will not remain in STALL indefinitely. Hence the two fairness constraints on the Request Agent can be guaranteed.

5.2.3 Guarantees on Processor1

The assumptions on outputs from Processor1 while verifying Request Agent of Processor0 have to be guaranteed. Processor1 interacts with Processor0 through the Request Agent. Hence, the assumptions can be guaranteed on Request Agent of Processor1. The external environment of Request Agent of Processor1 is identical to that of the Request agent of Processor0. The external environment of Request Agent of Processor0 has already been discussed in the first approach. The assumptions are specified as CTL properties which are to be verified on Processor1.

Guarantee on DRDY_O_1 **:**

In section 5.2.1.1, we made assumptions on output signal DRDY_O_1 of Processor1. After the Processor0 transaction is enqueued in the FIFO buffer of Processor1, it eventually goes to RESPONSE_PHASE . When the requested cacheline by the processor0 is present in Processor1(fifoWBInto = 1) and the Request type is EXT_MR or EXT_MRI, then DRDY_O_1 should be asserted by Processor1. This is verified on Request Agent of Processor0 by specifying it as a CTL property. In CTL language the property is stated as,

Property6:
AG((((processor.request_type = EXT_MR) + (processor1.request_type = EXT_MRI)
$\longrightarrow$ AF(DRDY_O_1 = 1));

The above formula passed on the Request Agent of Processor1 satisfying as guarantees, the assumptions made on DRDY_O_1.

5.2.4 Guarantees on Memory

Next, the assumptions on signals DRDY_O_M, TRDY and TRAN_OVER of Memory module need to be guaranteed. The assumptions on these signals has already been discussed in section 5.2.1.1. For this, Memory is the main machine. While the external environment includes Processor0 and Processor1. The signals from the external environment which are inputs to Memory have to be constrained. ADS_I which is a input to memory is a wired-ORed signal of ADS_O_0 and ADS_O_1. ADS_I can be asserted at any point of time. But it is sampled at the rising edge of the clock. Hence, ADS_I is non-determinised. If ADS_I = 1, then EXT_TRANS_REQ_I is non-determinised and can take any of the valid Request types. EXT_ADDR_I is also non-determinised and can take addresses mapping onto same cache lines e.g 0 and 2. When the transaction enters the SNOOP_PHASE, HIT_HITM_I is allowed to take any random value. A fairness constraint imposed on HIT_HITM_I to ensure that it is not equal to STALL indefinitely. The fairness constraint is an assumption we are making on the external environment. This assumption has already been shown to be guaranteed by the Cache unit as discussed in section 5.2.2.2.

***Guarantee on DRDY_O_M* :**

DRDY_O_0, DRDY_O_1 and DRDY_O_M are wired-ORed to generate DRDY_I, which goes as an input to Memory (Fig. 2.7). When the requested cacheline is present in another processor (Processor 0 or Processor 1), then DRDY_O_0 or DRDY_O_1 is asserted. For convenience, DRDY_O_0 and DRDY_O_1 is logically ORed to generate DRDY_O_C and assumed that when the requested cacheline is present in any of the Processors (HIT_HITM_I = 2'b01), then DRDY_O_C is asserted. This assumption has already been shown to be guaranteed by Processor1 in section 5.2.3. As Processor1 and Processor0 are identical, the assumption is guaranteed on the external environment (Processor0 and Processor1). When the transaction enters the RESPONSE_PHASE and the requested cache line is not present in either of the Processors, then DRDY_O_M is asserted, when the Request type is EXT_MR or EXT_MW. This can be verified on Memory module for CTL property stated as

Property7:

*AG(((fifoPhase0 = RESPONSÉ_PHASE) * ((request_type = EXT_MR) + (re-*

*quest_type = EXT_MRI)) * !(HIT_HITM_I = MODIFIED))*
    ⟶ AF(DRDY_O_M = 1));

Guarantee on TRDY :

TRDY is generated when the transaction is in RESPONSE_PHASE and when the requested cache line is in either of the processors and Request type is EXT_MR or EXT_MRI or when the transaction is in RESPONSE_PHASE and the Request type is EXT_MW. This can be verified on Memory block formally as

Property8:
AG ((fifoPhase0 = RESPONSE_PHASE) * ((request_type = EXT_MR) + (request_type = EXT_MRI)) * (HIT_HITM_I = MODIFIED)
    ⟶ AF(TRDY=1));

Guarantee on TRAN_OVER :

The assumptions in 5.2.1.1, should be guaranteed for TRAN_OVER. For a valid EXT_TRANS_REQ (!= NOREQ), TRAN_OVER should be asserted eventually. Formally,

Property9:
AG((EXT_TRANS_REQ != NOREQ) ⟶ AF(TRAN_OVER = 1));

All the above properties *passed* on the Memory block thereby satisfying as guarantees, all the assumptions made on the output signals of the Memory block.

Chapter 6

Results, Conclusion and Scope for future work

As discussed in Chapter 5, the verification of **liveliness** and **IOQ** property for the "general case" (where the opcodes of both the processors are kept variable) using the compositional verification technique known as the Assume-Guarantee approach, was successfully completed.

In this chapter we present the results. The chapter is organised as follows. Section 1 presents the results. Section 2 discusses the results and finally, section 3 points to future directions.

6.1 Results

The model size of the entire P6 bus system is given in the Table below (presented in chapter 3, but given here for convenience).

Design Parameters	Components
Combinational	5721
latches	298
edges	15114

6.1.1 First approach: Partitioning at the Processor level

In our first approach, we partitioned the entire system at the processor level. In this case, Processor0 is the main machine and the external environment consists of the Memory and Processor1 modules. The model size for this case is given as under:

Design Parameters	Components
Combinational	4321
latches	125
edges	11421

Table 6.1: **Model size in the first Approach**

The reachability analysis for this approach is given in Table 6.2:

Reachability analysis Parameters	Value
FSM depth	95
Reachable states	2.29×10^7
BDD size	2978111
analysis time (sec)	3182

Table 6.2: **Reachabilty analysis in first approach**

For this case, though full reachability analysis was possibl, it resulted in state explosion problem and therefore the verification could not be completed. This can be observed from the table where the BDD size and number of reachable states is enormous.

6.1.2 Second approach: Partitioning at the module level of Processor0

In our second approach, we partitioned the design at the module level of Processor0.

6.1.2.1 Request Agent:

In the first instance, we proceeded to verify the property on Request Agent of Processor0. In this case, the Request Agent of Processor0 is the main machine and the external environment consists of Processor1, Memory and combination of Execution unit and Cache module of Processor0.

Case 1: When the edges a,b are not removed in Fig. 5.1 and Fig. 5.4respectively.

The *model size* for this case is given in Table 6.3

Design Parameters	Components
Combinational	1184
latches	85
edges	2755

Table 6.3: **Model size of Request agent in second approach**

The *reachability analysis* is given in Table 6.4.

Reachability Analysis parameters	Value
FSM depth	31
Reachable states	158301
BDD size	61579
analysis time (sec)	29

Table 6.4: **Reachability analysis of request agent for case 1**

The following 2 properties(**liveliness** and **IOQ**) are verified on the Request Agent:

Property checked	passed/failed	Time(sec) taken to verify the property
Property3	passed	320 seconds
Property4	passed	55 seconds

Case 2: When the edges a, b are removed from the Fig. 5.1 and Fig. 5.4 respectively. The model size for case 2 is given in Table 6.5.
The reachability analysis is shown in Table 6.6.
The following two properties (**liveliness** and **IOQ**) are verified:

Design	Parameters
Combinational	1154
latches	85
edges	2681

Table 6.5: **Model size in case 2 of approach 2.**

Reachability analysis	value
FSM depth	18
reachable states	7106
BDD size	13920
analysis time (sec)	4

Table 6.6: **Reachability analysis in case 2 of second approach**

Property checked	status	Time(sec) taken to verify
Property3	passed	80
Property4	passed	30

6.1.2.2 Combination of Execution Unit and Cache

In this case, the main machine is the Execution unit + Cache of Processor0. The external environment is the Request Agent of Processor0.

The model size in this case is given in Table 6.7

Design Parameters	Components
Combinational	1346
latches	66
edges	3395

Table 6.7: **Model size when Execution unit + Cache as main machine**

The reachability analysis is given in Table 6.8.

CTL Properties:

The following 2 properties are verified on the the combination of Execution unit + Cache:

Reachability analysis	value
FSM depth	46
reachable states	6907
BDD size	3035
analysis time (sec)	3

Table 6.8: **Reachability analysis when Execution unit + Cache as main machine**

Property checked	status	time elapsed
Property1	passed	2 seconds
Property2	passed	3 seconds
Property5	passed	2 seconds

6.1.2.3 Request Agent of Processor1

The main machine is Request Agent of Processor1 and the external environment constitutes the Memory and Request Agent of Processor1. The model size and reachability analysis is same as that given in Table 6.3 and Table 6.4.

Property6 is verified on Request agent of processor1.

6.1.2.4 Memory

The main machine, in this case, is Memory. The external environment constitutes Processor0 and Processor1. The model size is given in Table 6.9.

Design Parameters	Components
Combinational	749
latches	49
edges	2094

Table 6.9: **Model size when Memory as main machine**

Reachability analysis is given in Table 6.10

The following 2 properties are verified :

Reachability analysis	value
FSM depth	11
reachable states	94
BDD size	449
analysis time (sec)	0

Table 6.10: **Reachability analysis when Memory as main machine**

Property	status	time elapsed(sec)
Property7	passed	0
Property8	passed	0
Property9	passed	0

6.2 Conclusion

In the first approach, even though the design is partitioned (i.e., Processor0, Processor1, Memory), we encountered state explosion problem was encountered for the properties for the "general" case. This can be attributed to the enormous state space generated by the product FSM of the Processor0 and the extracted FSM's of the external inputs. In the second approach, we identified partition boundaries in the module hierarchy and module interconnections based on our detailed analysis of the design. We took the help of state-transition graph in defining properties at the interface of these interacting modules, so as to decouple these modules and render the original verification problem into smaller verification problems involving only the individual decoupled modules.

With our approach, though it is difficult to measure the hidden cost based on the amount of designer input that is needed to partition the given model and the property into an appropriate set of assumptions and guarantees, it does enable a *Design for Verification* approach to be adopted. The final verification cost can be avoided, by employing smaller verification efforts on individual modules on individual modules, or groups of modules based on the designers detailed knowledge of the design. This approach can also exploit the natural evolution of an implementation based on both, top-down and bottom-up styles, and force the dessigner to adopt a design style which is more verification friendly.

Each of the above issues need detailed investigation based on the application of this approach to a wider class of industrial designs.

6.3 Scope for future work

Though we verified the liveliness and IOQ properties using Assume-Guarantee approach, it may not be the case for every design. These two properties involves the variables of a single (main) large machine within a system of interacting components. This enabled us to decompose the given global property into sub-properties belonging to ACTL, a subset of CTL. It may not always be possible to decompose a given global property into sub-properties belonging to ACTL. The cache coherance property which we have not verified, involves the variables of two interacting FSM's (cache of Processor0 and cache of Processor1). The verification of this property requires a different approach and this calls for future work. A good direction to investigate would be to find the rules by which a non-verifiable design instance can be transformed into a verifiable instance by modifying the behaviour of individual components and by changing the structure of interaction of these components.

Bibliography

[1] M. K. Ganai and I. M. Liu, “Formal verification of a Cache Coherance Protocol”, *A report submitted In University of Texas, Austin*, Dec. 1997.

[2] T. Kropf, *An introduction to Formal hardware Verification*, Springer-Verlag, Berlın.

[3] E. M. Clarke and J. M. Wing, “Formal Methods: State of the Art and Future directions”, *Technical report no. CMU-CS-96-178*, Carnegie Mellon University, Sep. 1996.

[4] C. J. Seger, “An Introduction to Formal Hardware Verification”, *Tech. Report, University of British Columbia, Canada*, 1992.

[5] G. V. Bochmann, “Hardware Specification with Temporal Logic: An Example”, *IEEE Transactions on Computers*, Vol. C-31, pp 223-230, March 1982.

[6] M. C. Browne, E. M. Clarke and D. L. Dill and B. Mishra, “Automatic Verification of Sequential Circuits using Temporal Logic”, *IEEE Transactions on Computers*, Vol. C-35, No. 12, pp 1035-1043, Dec. 1986.

[7] E. M. Clarke, E. A. Emerson and A. P. Sistla, “Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications”, *ACM Trans. on Programming Language and Systems*, Vol.8, No.2, pp 244-263, April 1986.

[8] K. L. McMillan, "Symbolic Model Checking", *Kluwer Academic Publishers*, 1993.

[9] The VIS Home page: *http://www-cad.eecs.berkeley.edu/~vis.*

[10] T. A. Henzinger, S. Qadeer and S. K. Rajamani, “You assume, We guarantee : Methodology and Case Studies”, *CAV98: Computer Aided Verification*, Lecture Notes in Computer Science, Springer-Verlag, pp. 440-451, 1998.

[11] E. M. Clarke, D. E. Long and K. L. McMillan, “A Language for Compositional Specification and Verification of Finite State Hardware controllers”, *Proceedings of the IEEE*, Vol. 79, pp.1283-1292, Sept. 1991.

[12] E. M. Clarke, D. E. Long and K. L. McMillan, “Compositional Model Checking”, *Proceedings of the Fourth Annual Symposium on logic in Computer Science*, pp.353-362, June 1989.

[13] M. Chiodo, M. Marelli, T. R. Shiple, A. L. Sangiovanni and R. K. Brayton, “Automatic Compositional Minimization in CTL Model Checking”, *URL: http://dev.acm.org/pubs/citations/proceedings/dac/304032/p172-chiodo/.*

[14] K. L. McMillan, "A compositional rule for hardware design refinement", *Computer Aided Verification (CAV97)*, O. Grumberg (Ed.), Haifa, Israel, pp. 24-35, 1997.

[15] S. K. Roy, H. Iwashita and T. Nakata, “Formal Verification of Pico Java External Bus Interface”, *Fujitsu preliminary Internal report*, January 1999.

[16] S. K. Roy, H. Iwashita and T. Nakata, “Formal Verification based on Assume and Guarantee Approach - A Case Study”, *Technical Report LTM-99-6805-001, Fujitsu Laboratories Limited, Kawasaki, Japan*, March 1999.

[17] D. E. Thomas and P. R. Moorby, *The Verilog Hardware Description Language*, Third edition, Kluwer Academic Publishers, Boston.

[18] N. Alexandridis, *Design of Microprocessor-Based Systems*, Prentice Hall International, Singapore.

[19] J. L. Heennessy and D. A. Patterson, *Computer Architecture - A quantitative Approach*, Morgan Kaufmann, San francisco.

[20] Thomas Shiple et al, "Formal Design Verification of Digital Systems", *URL: http://www.ece.utexas.edu/~adnan/.*

[21] "Tech Docs", *URL: http://www.resnet.uconn.edu/simjeff/techinfo/*, December 1997.

[22] A. Aziz, V. Singhal and R. K. Brayton, "Verifying Interacting Finite State Machines: Complexity Issues", *URL: http://www.ece.utexas.edu/~adnan/.*

Appendix

As an example, we give here, the verification procedure in VIS on Request agent of Processor0. We think that this will be of help to VIS beginners.

```
vis release 1.2 (compiled 24-Nov-00 at 6:44 PM)
vis> rlmv final_test.mv
    Warning: Some variables are unused in model PENTIUM_PRO_SYSTEM.
    Warning: Some variables are unused in model REQUEST_AGENT.
    Warning: Some variables are unused in model DRDY.
    Warning: Some variables are unused in model TRDY.
    Warning: Some variables are unused in model HITM.
    Warning: Some variables are unused in model TRANOVER.
vis> init_verify
    Warning: No checking for complete and deterministic specification.
    Warning: Do "help flatten_hierarchy" for detailed information.
vis> print_network_stats
    PENTIUM_PRO_SYSTEM combinational=1184 pi=0 po=0 latches=85 pseudo=9
const=101 edges=2755
vis> dvo -f sift
    Dynamic variable ordering forced with method sift....
vis>compute_reach -v 1
    Reachability analysis results:
    FSM depth = 31
    reachable states = 158301
    BDD size = 61579
    analysis time = 28
vis> time
    elapse: 29.0 seconds, total: 29.0 seconds
vis> mc -i ~/files_thesis/SNOOP/live.ctl
# MC: formula passed
AG((((REQUEST_AGENT1.EXT_TRANS_REQ_O<2>=1 +
REQUEST_AGENT1.EXT_TRANS_REQ_O<1>=1) +
REQUEST_AGENT1.EXT_TRANS_REQ_O<0>=1) —>
    AF(REQUEST_AGENT1.REQ_GRANTED=1)))
vis> time
    elapse: 229.2 seconds, total: 258.2 seconds
vis> print_fairness
    Fairness constraint:
    !((hit_hitm.HIT_HITM_O_1<0>=1 * hit_hitm.HIT_HITM_O_1<1>=1));
    !(cache.SNOOP_TAG_I<2>=1);
vis>reset_fairness
vis> time
    elapse: 0.0 seconds, total: 258.2 seconds
vis>mc -i ~/files_thesis/SNOOP/live.ctl
#MC: formula failed
AG((((REQUEST_AGENT1.EXT_TRANS_REQ_O<2>=1 +
REQUEST_AGENT1.EXT_TRANS_REQ_O<1>=1) +
REQUEST_AGENT1.EXT_TRANS_REQ_O<0>=1) —>
    AF(REQUEST_AGENT1.REQ_GRANTED=1)))
vis> time
    elapse: 66.1 seconds, total: 324.3 seconds
```

As seen in the example above, if we reset the fairness condition by *reset_fairness*, the CTL formula failed on the model.